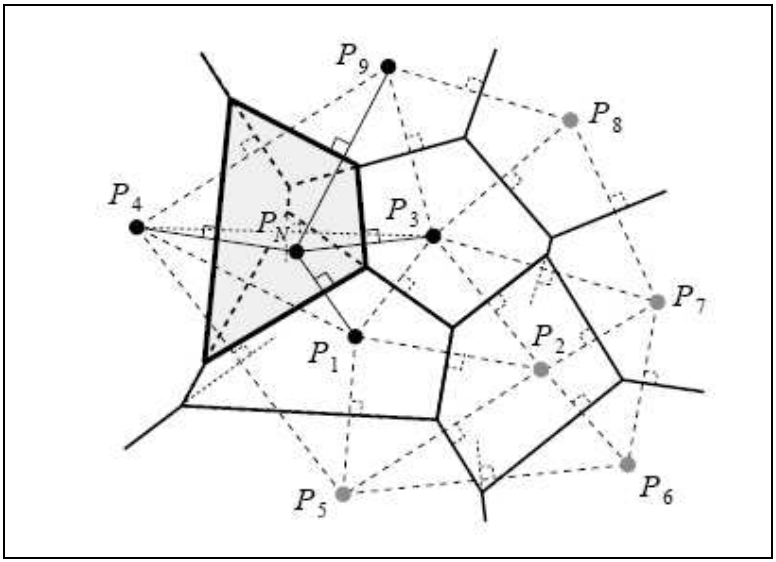
Boldea Costin-Radu

Algoritmică geometrică



2013

PREFAȚĂ

Calculatoarele sunt din ce în ce mai mult folosite pentru a rezolva pe scala largă probleme care sunt inerent geometrice. Obiectele geometrice cum ar fi punctele, liniile şi poligoanele formează bazele unei varietăți largi de aplicații importante în teoria algoritmilor.

Astfel se pot identifica mai multe domenii ale informaticii care se ocupă cu rezolvarea problemelor de natură geometrică, cum ar fi reprezentările grafice, procesarea imaginilor, robotica, fabricarea şi proiectarea asistată de calculator, dinamica fluidelor şi „geometria" bazelor de date, pentru a numi câteva dintre ele. Unul dintre scopurile Algoritmicii geometrice este de a oferi instrumentele de bază necesare din care programatorii pot să-şi construiască apoi aplicațiile. S-a făcut un progres semnificativ în acest sens, dar încă e departe de a fi pe deplin realizat.

Algoritmii geometrici sunt importanți de asemenea în studiul modelelor de analiză ale obiectelor fizice, începând de la clădiri şi automobile până la marea categorie a circuitelor integrate. Un proiectant care lucrează cu un obiect fizic are o intuiție geometrica care este greu de implementat în reprezentarea pe calculator a obiectului. Multe alte aplicații implică în mod direct prelucrarea datelor geometrice. De exemplu, o schema politica trucata pentru a diviza o regiune în zone care au populații egale ca număr (si care satisface alt criteriu cum ar fi localizarea majorității membrilor unui partid intr-o singura zona) constituie un algoritm geometric sofisticat. Alte aplicații abunda în matematica şi statistica, unde multe tipuri de probleme care apar pot fi încadrate firesc intr-o reprezentare geometrica.

Cursul îşi propune o introducere succintă în domeniul reprezentărilor grafice geometrice pe calculator, a problematicii şi metodelor sale, precum şi o abordare rapidă a domeniului geometriei computaționale.

Obiectivele cursului.

Cunoaşterea şi înțelegerea noțiunilor fundamentale de reprezentare grafică în C. Introducere în algoritmica geometrică. Prezentarea unor probleme fundamentale ale geometriei computaționale.

Necesar Hardware si software pentru desfasurarea cursului.

- Calculatoare personale cu OS Windows XP sau Windows 7
- Bloodshed Dev-C++ (utilitar gratuit) (http://www.bloodshed.net/devcpp.html) + pachetele grafice (http://www.cs.northwestern.edu/academics/courses/110/html/cs110-glutlib.html)

CUPRINS

CAPITOLUL I. INTRODUCERE ÎN ALGORITMICA GEOMETRICĂ 9

1.1 CE ESTE ALGORITMICA GEOMETRICĂ? 9
1.2 LIMITELE GEOMETRIEI COMPUTAȚIONALE 11
1.3 EXEMPLE TIPICE DE PROBLEME ALE GEOMETRIEI COMPUTAȚIONALE 13
1.4 TEME LABORATOR/SEMINAR 15

CAPITOLUL II. GRAFICA ÎN LIMBAJULUI C 16

2.1 CONFIGURAREA MODULUI GRAFIC ÎN DEV C++ 16
2.2 FUNCȚIILE MODULUI GRAFIC ÎN DEV C++ 22
2.2.1. Inițializarea ecranului grafic în Dev C++ 22
2.2.2. Culorile modului grafic în Dev C++ 23
2.2.3. Coordonate-ecran ale pixelilor în Dev C++ 25
2.2.4. Funcțiile elementare de desenare 27
2.2.5. Ecran virtual si ecran grafic 30
2.3 TEME LABORATOR/SEMINAR 33

CAPITOLUL III. NOȚIUNI GEOMETRICE ELEMENTARE 34

3.1 PUNCTE ȘI VECTORI 34
3.2 GEOMETRIE AFINĂ ȘI EUCLIDIANĂ 37
3.2.1 Operații afine cu puncte, vectori şi scalari 37
3.2.2. Produsul scalar 39
3.3 DREPTE ȘI CERCURI ÎN PLAN 40
3.4 REPREZENTAREA SPAȚIULUI 3D 42
3.4.1 Sistemul obiect şi observator 42
3.4.2 Sisteme de coordonate 3D 43
3.5 DREPTE, PLANURI ȘI SFERE ÎN SPAȚIU 45
3.6. REPREZENTAREA SPAȚIULUI 3D PE ECRAN 46
3.7. TEME DE LABORATOR 49

CAPITOLUL IV. TRANSFORMĂRI GEOMETRICE 51

4.1 TRANSFORMĂRI GEOMETRICE ELEMENTARE ÎN PLAN 51
4.1.1 Translația 52
4.1.2 Scalarea 52
4.1.3 Simetria 53
4.1.4 Rotația 54

4.2 TRANSFORMĂRI GEOMETRICE ELEMENTARE ÎN SPAȚIU 55
4.2.1 Translația *55*
4.2.2 Scalarea *56*
4.2.3 Rotația *57*
4.2.4 Concatenarea şi transformarea inversă *58*
4.3. TRANSFORMAREA DE PERSPECTIVĂ 60
3.7. MODELUL CAMEREI DE LUAT VEDERI 64
4.4. TEME DE LABORATOR 66

CAPITOLUL V. ALGORITMI ELEMENTARI DE GEOMETRIE COMPUTAȚIONALĂ 71

5.1 PUNCTE, LINII ŞI POLIGOANE 71
5.2 ORIENTAREA TRIUNGIURILOR ȘI TESTUL DE COLINIARITATE 74
5.2.1 Orientarea punctelor *74*
5.2.2 Suprafețe şi unghiuri *76*
5.3 INTERSECȚII PROPRII ȘI IMPROPRII DE SEGMENTE 77
5.4 PROBLEMA DRUMULUI SIMPLU ÎNCHIS 80
5.5 APARTENENȚA LA INTERIORUL UNUI POLIGON 82
5.6. TEME DE LABORATOR 86

CAPITOLUL VI. PROBLEME DE INTERSECȚII 92

6.1. INTERSECȚII GEOMETRICE 93
6.2. INTERSECȚII DE SEGMENTE DE DREAPTĂ 94
6.2.1 Intersecția brută a segmentelor de dreaptă *95*
6.2.2 Linii orizontale si verticale *96*
6.2.3 Intersecții de segmente arbitrare *98*
6.3. TEME DE LABORATOR 100

CAPITOLUL VII. ÎNFĂŞURĂTOAREA CONVEXĂ 105

7.1 PROBLEMA DETERMINĂRII ACOPERIRII CONVEXE ȘI ALGORITMUL NAIV 105
7.2 METODA ÎMPACHETĂRII (WRAPPING) 108
7.3 SCANAREA GRAHAM 113
7.4 METODA RAPIDĂ PRIN SELECȚIE (*QUICKHULL*) 118
7.5. TEME DE LABORATOR 120

CAPITOLUL VII. SUBDOMENII ALE UNUI POLIGON 126

8.1. PROBLEMA NUCLEULUI 126
8.2. TRIANGULAREA POLIGOANELOR 129
8.2.1 Triangularea poligoanelor convexe *131*
8.2.2 Triangularea poligoanelor simple arbitrare *134*

8.3. TEME DE LABORATOR 139

CAPITOLUL IX. DIAGRAME VORONOI ŞI TRIANGULĂRI DELAUNAY 140

9.1. PROBLEMA DIAGRAMELOR VORONOI 140
9.2. CONSTRUCȚIA ALGORITMICĂ A DIAGRAMELOR VORONOI 142
9.3. TRIANGULAREA DELAUNAY A UNUI ANSAMBLU DE PUNCTE 143
9.4. ALGORITM DE DETERMINARE A TRIANGULĂRII DELAUNAY ŞI A DIAGRAMEI VORONOI 145
9.6. TEME DE LABORATOR 151

CAPITOLUL X. PROBLEMA CELUI MAI SCURT DRUM ÎNTR-UN CÂMP CU OBSTACOLE ŞI GRAFUL DE VIZIBILITATE 152

10.1. PROBLEMA CELUI MAI SCURT DRUM ÎNTR-UN CÂMP CU OBSTACOLE 152
10.2. GRAFUL DE VIZIBILITATE 153
10.3. TEMĂ DE LABORATOR 157

BIBLIOGRAFIE 158

Capitolul I. Introducere în Algoritmica geometrică

Descriere Generala

Primul capitol este introductiv, propunându-şi familiarizarea cu problematica şi obiectivele algoritmicii geometrice şi a reprezentărilor grafice pe calculator.

Obiective

-Definirea domeniului şi a obiectului de studiu al geometriei computaţionale, ca subdomeniu al algoritmicii

-Familiarizarea cu problematica geometriei computaţionale

1.1 Ce este algoritmica geometrică?

Problemele de geometrie sunt uşor de vizualizat, ceea ce reprezintă dintr-un anumit punct de vedere un handicap. Multe probleme care pot fi rezolvate instantaneu de către o persoană care se uită la o bucată de hârtie (de exemplu: apartenenţa unui punct dat la interiorul unui poligon dat) necesită programe care nu sunt uşoare. Pentru problemele mai complicate, ca şi pentru multe alte aplicaţii, metoda de rezolvare *convenabila* prin intermediul unui calculator poate fi destul de diferită faţă de metoda de rezolvare folosită de către o persoană fizică.

Se presupune în mod uzual că algoritmii geometrici au o istorie lungă datorită naturii constructive a geometriei vechi şi pentru că unele aplicaţii utile sunt atât de larg răspândite, dar de fapt cea mai mare parte din munca depusă în acest domeniu este destul de recentă. Bineînţeles, de multe ori munca vechilor matematicieni este foarte utila în dezvoltarea algoritmilor pentru computerele moderne.

Termenul de **Geometrie computaţională** a fost utilizat prima dată de Marvin Minsky în cartea sa "Perceptroni", care se referea la recunoaşterea formelor (*pattern recognition*) şi de asemenea a fost des folosit la descrierea algoritmilor pentru manipularea curbelor şi suprafeţelor. Totuşi, în mare măsură utilizarea sa curentă este asociată *subdomeniului teoriei algoritmilor*

care studiază proiectarea şi analiza algoritmilor eficienţi pentru probleme derivate din geometrie. În acest sens am preferat să folosim termenul de **Algoritmică geometrică** pentru acest domeniu, mai potrivit ca descriere decăt „geometrie computaţională"

Măsurarea calităţii unui algoritm în geometria computaţională se face tradiţional prin *timpul asimptotic de rulare al celui mai rău caz*. Astfel, un algoritm care funcţionează în timp $O(n)$ este mai bun decât unul care funcţionează în timp $O(n \log n)$ care este mai bun decât unul care funcţionează în timp $O(n^2)$. (Problema particulară de mai sus poate fi rezolvată în timp $O(n^2 \log n)$ de un algoritm relativ simplu, în $O(n \log n)$ de un algoritm relativ complex şi poate fi aproximat destul de bine de un algoritm mai simplu al cărui timp de rulare este de asemenea $O(n \log n)$). În unele cazuri se ia în considerare *cazul timpului mediu* de funcţionare. Totuşi pentru multe tipuri de intrări geometrice este dificil să se definească intrări distributive deoarece ambele sunt uşor de analizat şi reprezentative pentru intrările tipice.

Domeniul geometriei computaţionale s-a dezvoltat rapid la sfârşitul anilor '70, în anii '80 şi '90, şi continuă să se dezvolte. Istoric, geometria computaţionale s-a dezvoltat ca o generalizare a studiului algoritmilor pentru sortare (*sort*) şi căutare (*searching*) într-un spaţiu unidimensional spre probleme care implică intrări multidimensionale. De asemenea, în anumită măsură, s-a dezvoltat ca o limită a teoriei de calcul a graficii, studiind pattern-urile care se nasc normal din proprietăţile geometrice.

Datorită istoriei sale, domeniul geometriei computaţionale s-a concentrat în mare parte pe probleme în spaţiul bidimensional şi s-a extins mai puţin în spaţiul tridimensional. Când problemele sunt studiate în spaţii multidimensionale, de obicei se presupune că dimensiunea spaţiului este o constantă mică (să spunem 10 sau mai puţin). Pentru că domeniul a fost dezvoltat de cercetători a căror pregătire era în algoritmi discreţi (în opoziţie cu analiza numerică)

domeniul s-a concentrat deasemenea mai mult pe natura discretă a problemelor de geometrie în opoziție cu problemele continue. *Geometria computațională se ocupă în primul rând cu forme drepte sau plane (linii, segmente de linie, poligoane, planuri şi poliedre) sau forme simple curbate cum sunt cercurile*. Acesta este în contrast, să spunem, cu domenii cum ar fi modelarea în spațiu, care se concentrează pe probleme care studiază curbe şi suprafețe complexe.

1.2 Limitele geometriei computaționale

Există câteva motive relativ normale din cauza cărora geometria computațională nu se va adresa în totalitate amatorilor de geometrie, şi aceste limite ar trebui bine înțelese de la început.

Una dintre acestea este *natura discretă* a geometriei computaționale. Într-un anumit sens orice problemă care este rezolvată pe calculatoare digitale trebuie exprimată în formă discretă, dar multe domenii de aplicație tratează probleme definite pe un spectru mai larg, de la aproximații discrete până la fenomene continue. De exemplu, la procesarea imaginii, imaginea poate fi o discretizare a unei funcții bidimensioanale continue în nuanțe de gri, iar, în robotică, oscilația în sistemele dinamice de control are caracter continuu. Cu toate acestea, există multe aplicații în care obiectele sunt de natură discretă. De exemplu, în sistemele geografice de date, rețelele rutiere sunt discretizate în colecții de segmente de linie.

O altă limită este impusă de faptul că geometria computațională se ocupă în primul rând de *forme drepte sau plane*. Într-un context mai larg, acesta este rezultatul faptului că cercetătorii din geometria computațională nu erau pregătiți în geometrie, ci în programarea algoritmilor discreți. Astfel ei au ales probleme în care geometria şi calculul numeric joacă un rol relativ mic. O mare parte din modelarea în spațiu, dinamica fluidelor şi robotică, se ocupă cu forme care sunt modelate cu suprafețe curbate. Totuşi este posibil să se aproximeze forme curbe cu poligoane plane sau poliedre.

Această presupunere a îngăduit geometriei computaționale să se ocupe de elementele combinatorice ale multora dintre probleme, în opoziție cu preocuparea față de problemele numerice.

Încă o limită este aceea că geometria computațională s-a concentrat în primul rând pe *probleme bidimensionale* şi într-o arie mai restrânsă pe probleme tridimensionale. „Aspectul" mai plăcut al problemelor bidimensionale este că sunt uşor de vizualizat şi uşor de înțeles. Dar multe dintre problemele concrete se plasează contextual în spațiul tridimensional sau spații cu dimensiuni mai mari.

În ciuda acestor limite există încă o arie remarcabilă de probleme interesante cărora geometria computațională li se adresează cu succes. De-a lungul anilor '80 domeniul a dezvoltat multe tehnici pentru proiectarea eficientă a algoritmilor geometrici. Acestea includ bine-cunoscutele metode cum ar fi divide şi vei cucerii şi programarea dinamică, împreună cu câteva metode nou descoperite care par a fi potrivite şi specifice algoritmilor geometrici. O mare parte a acestui curs se va baza pe înțelegerea tehnicii de proiectare eficientă a algoritmilor geometrici.

Între anii 1980 şi începutul anilor 2000 multe dintre problemele deschise ale geometriei computaționale au fost rezolvate în sensul că, teoretic, s-au dezvoltat algoritmi optimi pentru ele. O bună parte dintre aceşti algoritmi au fost greu de implementat din cauza complexității algoritmilor şi structurilor de date pe care le necesitau. Mai mult, aplicațiile care existau erau de multe ori sensibile la erori geometrice care le determinau să producă rezultate greşite sau la blocaje. De exemplu, un programator poate concepe un algoritm pentru determinarea intersecției unei mulțimi de segmente de dreaptă fără a lua în considerație situația când trei segmente de dreaptă se intersectează într-un singur punct. În aceasta situație rară, structura de date folosită poate fi deficientă şi algoritmul se opreşte.

Multe dintre cercetările recente în geometria computațională s-au axat pe încercarea de a face accesibile practicienilor rezultatele teoretice ale algoritmicii geometrice. Acest lucru s-a făcut prin

simplificarea algoritmilor existenți şi producerea de biblioteci întregi de proceduri predefinite.

1.3 Exemple tipice de probleme ale geometriei computaționale

a) **Acoperiri convexe:** Convexitatea este o proprietate geometrică foarte importantă. O mulțime geometrică este *convexă* dacă pentru fiecare două puncte din mulțime, segmentul de dreaptă care le unește face parte din mulțime. Una dintre primele probleme identificate în domeniul geometriei computaționale este aceea de a calcula forma convexă cea mai mică, numită *acoperire convexă*, care „acoperă" o mulțime de puncte date.

b) **Intersecții:** Una dintre problemele geometrice de bază este aceea de a determina când două mulțimi de obiecte intersectează o alta. A determina dacă obiectele complexe se intersectează se reduce de cele mai multe ori la a determina care dintre perechile individuale de identități primare (de exemplu: segmente de dreaptă) se intersectează.

c) **Triangularea şi partiționarea:** Triangularea este caracteristică celor mai generale probleme de subdivizare a unui domeniu complex într-o colecție de domenii "simple", cu largi aplicații în modelare în inginerie şi în geodezie.

d) **Programarea liniară în dimensiuni reduse (2,3):** Multe probleme de optimizare în geometria de calcul pot fi declarate de forma unei probleme de programare liniară, şi anume: să se determine punctele extreme (ex. cele mai ridicate sau cele mai scăzute) care satisfac un set de inegalități liniare. Programarea liniară este o problemă importantă în optimizarea combinatorică, şi de cele mai multe ori oamenii trebuie adesea să rezolve

astfel de probleme în spații cu sute sau poate mii de dimensiuni. Totuşi sunt multe cazuri interesante în spații cu număr mic de dimensiuni, dar cu foarte multe constrângeri (exprimate ca inegalități) unde există soluții eficiente foarte simple.

e) **Aranjamente de drepte şi dualitate**: Poate una dintre cele mai importante structuri matematice în geometria computațională este aceea a aranjamentului de drepte (sau la modul general aranjamentul de curbe şi suprafețe). Se dau *n* linii într-un plan, aranjamentul este doar graficul format prin considerarea punctelor de intersecție ca margini şi segmentele de dreaptă care le unesc ca verticale. Se poate arăta că o astfel de structură poate fi construită în timp $O(n^2)$ [1].

f) **Diagramele Voronoi şi triunghiurile Delaunay:** Find dată o mulțime *S* de puncte în spațiu, una dintre cele mai importante probleme este problema celui mai apropiat vecin. Considerând un punct care nu aparține lui *S*, care punct din *S* este cel mai apropiat lui? Una dintre tehnicile folosite pentru rezolvarea acestei probleme este aceea de a subdiviza spațiul în regiuni, potrivit distanțelor la cel mai apropiat punct. Această partiție geometrică a spațiului este numită *diagramă Voronoi*. Structura duală, numită *triungulația Delaunay*, are deasemenea multe proprietăți interesante, fiind cea mai practică triangulare utilizabilă în geodezie şi domeniul măsurătorilor de teren (cadastru), din cauza erorilor mici pe care le introduce.

g) **Regăsirea rapidă a informației (geometrice):** Problemele geometrice de căutare (*searching*) au urmatoarea formă generală: se dă o mulțime de date (de exemplu puncte, drepte, poligoane) care nu se pot modifica, se cere preprocesarea acestei mulțimi de date într-o structură de date astfel încat la un anumit tip de întrebare să se poată răspunde cât mai eficient posibil. De exemplu, o problemă de *căutare* a *celui mai apropiat vecin* este: determinarea punctul din mulțimea de date care este cel mai apropiat de un punct dat. O problemă *de*

poziționare este: determinarea mulțimii de puncte (sau calculați numărul de puncte) din mulțimea de date care sunt poziționate într-o anumită regiune. Regiunea poate fi un dreptunghi, un disc, sau o formă poligonală.

h) **Determinarea celui mai scurt drum într-un câmp cu obstacole** este poate una dintre cele mai importante probleme de robotică din ultimii ani. Evitarea obstacolelor de către un robot în condiții de totală autonomie a devenit celebră ca problemă odată cu lansarea, de către NASA, în urmă cu câțiva ani, a primului robot pe Marte (misiunea *Marspathfinder*) şi blocarea acestuia, timp de mai multe ore, în fața unui simplu bolovan.

Rezumat: Geometria computațională se ocupă cu studiul metodelor de rezolvare algoritmică a unor probleme cu caracter geometric, de implementarea algoritmilor rezultanți şi, de asemenea, de optimizarea lor.

1.4 Teme Laborator/Seminar

1. Descrieți pe scurt o problemă de geometrie care poate fi abordată algoritmic, diferită de cele expuse mai sus.
2. Identificați eventuale conexiuni între geometria computațională şi domeniul reprezentărilor grafice pe calculator.

Capitolul II. Grafica în limbajului C

Descriere Generala

Limbajul C permite accesul la o biblioteca de funcţii grafice care are peste 70 de funcţii grafice, de la funcţii de nivel înalt (setviewport(), bar3d(), drawpoly(), etc.) până la funcţii orientate pe bit (getimage(), putimage(), etc.), incluse toate în biblioteca de funcţii grafice graphics.h. Biblioteca grafică conţine numeroase stiluri de linii şi modele de umplere, câteva fonturi cărora li se poate modifica dimensiunea, care pot fi aliniate şi pot fi orientate orizontal sau vertical.

Funcţiile bibliotecii grafice pot fi clasificate în următoarele categorii:

1) funcţii de definire a sesiunii de desen;
2) funcţii pentru desenarea de texte;
3) funcţii de trasare pentru primitive geometrice;
4) funcţii la nivel de pixel;
5) funcţii pentru determinarea contextului curent al desenului.

În acest curs am prezentat varianta de implementare a bibliotecii grafice asociată compilatorului Bloodshed Dev-C++

Obiective

1. Iniţializarea modului grafic în Dev C++.
2. Introducerea funcţiilor de bază pentru controlul ecranului şi pentru reprezentări grafice.

2.1 Configurarea modului grafic în Dev C++

Modul grafic presupune că ecranul este format din ***pixeli***, puncte luminoase de diverse culori dispuse în rânduri şi coloane (sub formă de matrice). Numărul acestora depinde de adaptorul grafic şi

se numeşte ***rezoluție***. O rezoluție este cu atât mai bună cu cât este mai mare. De exemplu, adaptorul EGA oferea o rezoluție de 640 de coloane şi 350 de rânduri, adaptorul VGA oferă o rezoluție de 480 de rânduri şi 640 de coloane sau chiar mai mare, de până la 768 de rânduri cu 1024 de coloane (standard SVGA). În prezent rezoluția grafică este limitată doar de caracteristicile tehnice ale monitorului.

În Dev C++, modul grafic este definit în cadrul unei ferestre (*window*) de lucru specială, în care desenarea şi acțiunea *mouse*-ului sunt posibile. Pentru utilizarea graficii în Dev C++, este necesară mai întâi parcurgerea unor etape de configurare a compilatorului. În cele ce urmează am presupus că dispunem de utilitarul Bloodshed Dev-C++ deja instalat. În cazul în care nu dispuneți de acesta, un kit de instalare poate fi descărcat gratuit de la adresa http://www.bloodshed.net/devcpp.html . Pacheteul grafic presupune instalarea a două fişiere suplimentare **graphics.h** şi **libbgi.a**. (http://www.cs.northwestern.edu/academics/courses/110/html/cs110-glutlib.html)

În continuare am prezentat pe scurt principalele etape de configurare a utilitarului Dev C++ pentru a utiliza elementele de grafică din pachetul grafic.

Pasul 1: Configurarea linker-ului Dev-C++.

Modificarea setărilor din cadrul panoului de control al compilatorului

- Din meniul "`Tools`" se selectează "`Compiler Options`".
- În panoul "`Settings`" se selectează "Linker" pe coloana din dreapta şi se modifică opțiunea "`Generate debugging information`" în "`Yes`"

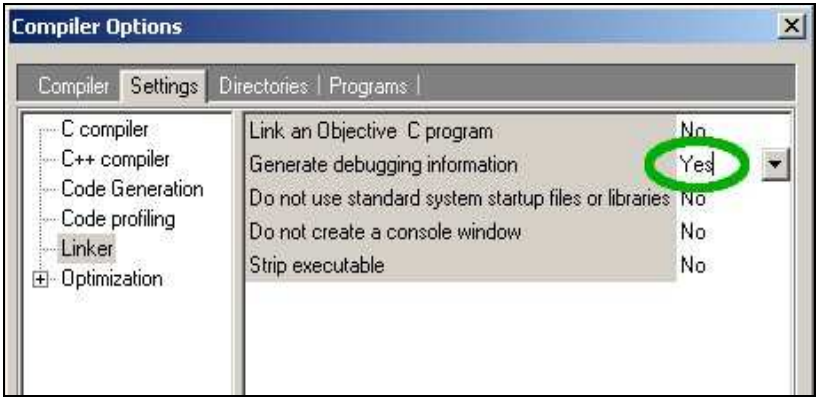


Fig.2.1 Configurarea opțiunii de debug

Pasul 2: Crearea unui proiect nou

Un proiect poate fi considerat ca un „dosar” în care sunt stocate toate elementele necesare compilării unui program. Un program în Dev C++ care conține elemente grafice nu poate fi compilat direct, el necesitând adiționarea bibliotecilor grafice.

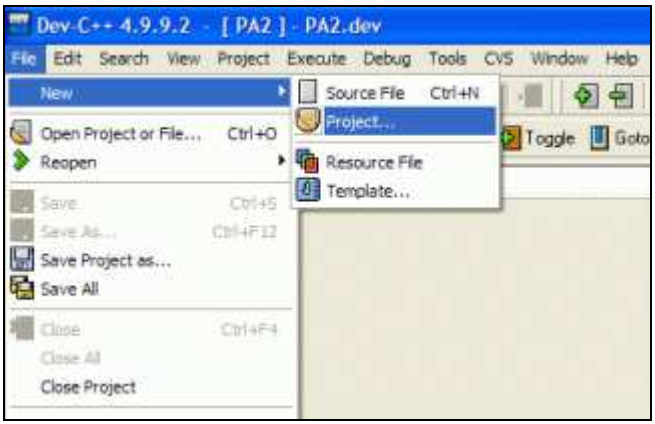


Fig.2.2 Crearea unui proiect nou

- Din meniul "`File`" se selectează "`New`", "`Project...`".
- Se alege opțiunea "`Empty Project`". În această etapă programul cere introducerea unui nume pentru noul proiect, nume care va fi păstrat de asemenea de către executabilul final.
- De asemenea este util de salvat proiectul într-un director nou, unde utilitarul Dev C să poată salva restul fişierelor adiționale ale proiectului

Pasul 3: Introducerea de fişiere sursă.

Fişierele sursă se pot introduce în două moduri:

- Din meniul "`File`" folosind opțiunea "`New Source File`" (CTRL+N) , sau
- Din meniul "`Project`" selectând "`New File`".
  De notat că utilitarul Dev-C++ nu cere un nume pentru noile fişiere sursă până când nu încercați să:
  1. compilați proiectul
  2. salvați proiectul
  3. salvați explicit fişierul sursă din meniul principal
  4. sau să ieşiți din Dev-C++

Pasul 4. Introducerea bibliotecii grafice (graphics.h)?

Clasicul emulator BGI utilizat de versiunile de Borland C fiind inapropiat pentru Dev C++, avem nevoie de librăria WinBGIm produsă de Michael Main.

Fişierele de care avem nevoie sunt:

graphics.h (care se introduce în C:\Dev-Cpp\include)

libbgi.a (care se introduce în C:\Dev-Cpp\lib)

(http://www.cs.northwestern.edu/academics/courses/110/html/cs110-glutlib.html)

Bibliotecile grafice pot fi utilizate doar dacă sunt specificate explicit în parametrii de compilare. Astfel:

- Din meniul "Project" se alege"Project Options" (ALT+P).
- Se alege tab-ul "Parameters"
- În coloana "Linker", se introduce textul:

```
-lbgi
-lgdi32
-lcomdlg32
-luuid
-loleaut32
-lole32
```

Project Options -> Parameters:

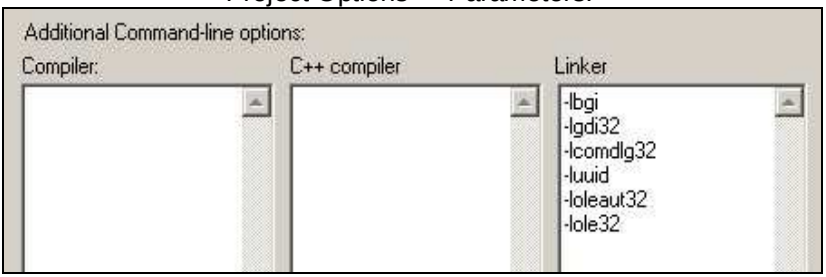


Fig.2.3 Configurarea linker-ului

- Clic "OK" în final.

Acum utilitarul Dev C++ este configurat să suporte funcțiile grafice. Testarea funcționării corecte a compilatorului se poate face simplu introducând următorul program scurt:

```
#include <graphics.h>
  int main()
  {
      initwindow(500,300); //deschide o fereatră
grafică
      moveto(50,50);
```

```
      lineto(150,250);
      while(!kbhit());      //aşteaptă apăsarea
unei taste
      closegraph();         //închide ferastra
grafică
      return 0;
 }
```

Compilarea şi execuția acestui program ar trebui să aibe ca efect crearea unei ferestre noi de dimensiune 500x300 de puncte (pixeli) și desenarea unei linii de la poziția (50,50) la poziția (150,250) relativ la colțul stânga sus al ferestrei.

2.2 Funcțiile modului grafic în Dev C++

2.2.1. Inițializarea ecranului grafic în Dev C++

Limbajul C are peste 70 de funcții grafice, de la funcții de nivel înalt (setviewport(), bar3d(), drawpoly(), etc.) până la funcții orientate pe bit (getimage(), putimage(), etc.), incluse toate în biblioteca de funcții grafice graphics.h. Biblioteca grafică conține numeroase stiluri de linii şi modele de umplere, câteva fonturi cărora li se poate modifica dimensiunea, care pot fi aliniate şi pot fi orientate orizontal sau vertical. De asemenea, pachetul grafic include drivere pentru perifericele grafice (*.BGI) şi fişiere de fonturi (*.CHR). Pentru a putea folosi funcțiile grafice trebuie ca fiecare fişier care foloseşte grafică să conțină directiva:

```
#include <graphics.h>
```

Pentru a inițializa modul grafic, trebuie mai întâi apelată funcția **initwindow,** care identifică adaptorul grafic, încărcă şi inițializează cel mai potrivit driver, trece sistemul în modul grafic şi întoarce controlul rutinelor grafice. Putem cere funcției **initgraph** să folosească un anumit driver grafic şi un anumit mod, sau să autodetecteze adaptorul grafic şi să aleagă driverul corespunzător.

Funcția **initwindow** resetează toate setările grafice la valorile lor implicite (poziția curentă, paleta, culoarea, viewport-ul etc.), resetează codul de eroare la 0 şi încarcă driverul grafic alocând memorie pentru el.

Această funcție poate fi folosită singură sau împreună cu o altă funcție numită **detectgraph** care determină parametri adaptorului grafic. Prototipul ei este:

```
void initwindow(int x1,int y1,char* nume,int
                x2,int y2);
```

unde (x1,y1) reprezintă dimensiunile ferestrei grafice, date în pixeli, *nume* este numele ferestrei, care va apare explicit pe bara albastră superioară de control a poziției ferestrei (dacă lipseşte *nume*, valoarea implicită utilizată este "Win BGI"), iar (x2,y2) este poziția relativă a colțului stânga sus al ferestrei față de colțul stânga sus al ecranului.

Din modul grafic se poate ieşi apelând funcția **closegraph** de prototip:

```
void closegraph(void);
```

2.2.2. Culorile modului grafic în Dev C++

Adaptoarele grafice sunt prevăzute cu o zonă de memorie în care se păstrează date specifice gestiunii ecranului. Acestă zonă de memorie poartă numele de *memorie video*.

În cazul adaptoarelor color, unui pixel îi corespunde o culoare. Culoarea pixelilor se păstrează pe biți în memoria video. Memoria video necesară pentru a păstra starea ecranului setat în mod grafic, se numeşte *pagină video*. Gestiunea culorilor este dependentă de tipul de adaptor grafic existent la microprocesor.

În cele ce urmează considerăm adaptoarele grafice de tip EGA/VGA. Numărul maxim al culorilor care pot fi afişate cu ajutorul unui adaptor EGA este de 64. Culorile se codifică prin numerele întregi din intervalul [0,63]. Cele 64 de culori nu pot fi definite afişate simultan pe ecran.

În cazul adaptorului EGA se pot afişa simultan pe ecran cel mult 16 culori. Culorile se codifică prin numerele întregi din intervalul [0,15]. Mulțimea culorilor care pot fi afişate pe ecran simultan se numeşte *paletă*. Culorile din componența unei palete pot fi modificate de utilizator prin intermediul funcțiilor standard. La inițializarea modului grafic se setează o paletă implicită. Paleta se defineşte cu ajutorul unui tablou de 16 elemente pentru adaptorul

EGA. Elementele acestui tablou au valori din intervalul [0,63], unde fiecare element din acest tablou reprezintă codul unei culori. Codurile culorilor din paleta implicită au denumiri simbolice definite în fişierul **graphics.h**.
Funcţiile de gestiune a culorilor pot avea ca parametri nu numai codurile culorilor, ci şi indecşi în tabloul care defineşte culorile unei palete. De aceea, indicii din intervalul [0,15] pot fi referiţi prin constante simbolice definite în fişierul **graphics.h**. Aceste denumiri sugerează culoarea din compunerea paletei.

Culoarea fondului de ecran (**background**) este întotdeauna cea corespunzătoare indicelui 0. Culoarea pentru desenare (**foreground**) este cea corespunzătoare indicelui 15.

Culoarea de fond poate fi modificată cu ajutorul funcţiei **setbkcolor**. Aceasta are prototipul:

```
void setbkcolor(int culoare);
```

unde: `culoare` este index în tabloul care defineşte paleta. De exemplu, dacă se utilizează apelul:

```
setbkcolor(BLUE);
```

atunci culoarea de fond a textului ce urmează a fi scris pe ecran devine albastră. Pentru a schimba culoarea întregul ecran de desenare se foloseşte instrucţiunea suplimentară

```
void cleardevice( );
```

la începutul sesiunii de lucru:

```
setbkcolor(BLUE);
     cleardevice();
```

Pentru a cunoaşte culoarea de fond curentă se poate apela funcţia **getbkcolor** de prototip:

```
int getbkcolor(void);
```

Ea returnează indexul în tabloul care defineşte paleta pentru culoarea de fond.

Culoarea pentru desenare poate fi modificată folosind funcţia **setcolor** de prototip:

```
void setcolor(int culoare);
```

unde: `culoare` este indexul unei culori în tabloul care defineşte paleta.

În sfârşit, afişarea şirurilor de caractere în modul grafic diferă de afişarea normală din C (cu `printf()`) sau C++ (cu `cout`). Pentru afişarea unui mesaj *c* începând din locația punctului se utilizează:

```
void outtextxy(int x,int y, *char c)ş
```

unde (*x,y*) sunt coordonatele de la care se afişează stringul *c*. Pentru afişarea numerelor se va folosi o conversie a acestora în şiruri de caractere, de exemplu prin funcția

```
*char itoc(int n).
```

În tabelul următor se indică codurile culorilor pentru paleta implicită.

Culorile în C	
denumire simbolică	Valoare
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

2.2.3. Coordonate-ecran ale pixelilor în Dev C++

În mod grafic, ecranul se compune din *n*m* pixeli, în funcție de rezoluția adaptorului grafic. Aceasta înseamnă că pe ecran se pot

afişa *m* linii a cate *n* pixeli fiecare. Poziția unui pixel se defineşte printr-o pereche ordonată de numere întregi: (x,y) numite coordonatele pixelului. Coordonata x defineşte coloana pixelului, iar y defineşte linia acestuia. Pixelul aflat în colțul din stânga sus are coordonatele (0,0). Coloanele se numerotează de la stânga spre dreapta, iar liniile de sus în jos.

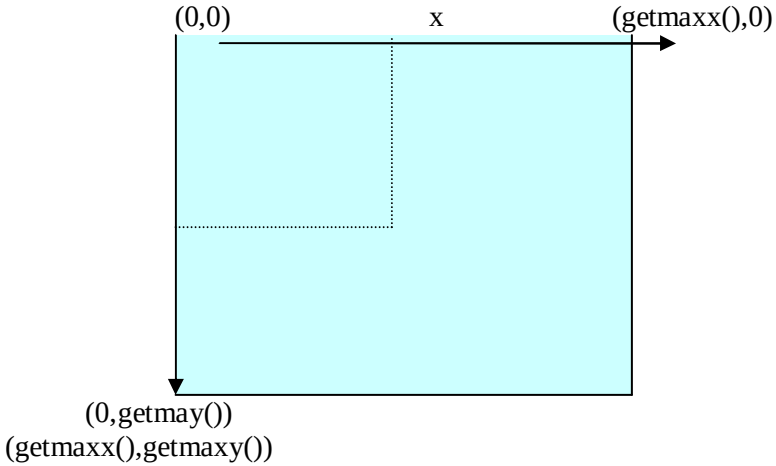


Fig.2.4 Ecranul grafic în C

Biblioteca grafică a programului Dev C++ conține funcții care permit utilizatorului să obțină următoarele informații relativ la ecran:

- coordonata maximă pe orizontală;
- coordonata maximă pe verticală;
- poziția curentă (pixel curent).

Prototipurile acestor funcții sunt:

```
int   getmaxx(void);
```
funcția returnează coordonata maximă pe orizontală (abscisa maximă);

```
int   getmaxy(void);
```
funcția returnează coordonata maximă pe verticală (ordonata maximă);

```
int   getx(void);
```

funcția returnează poziția orizontală (abscisa) a pixelului curent;

```
int   gety(void);
```

funcția returnează poziția verticală (ordonata) a pixelului curent.

2.2.4. Funcțiile elementare de desenare

Biblioteca standard a sistemului pune la dispoziția utilizatorului o serie de funcții care permit desenarea şi colorarea unor figuri geometric

Un punct colorat (pixel) se afişează cu ajutorul funcției **putpixel** de prototip:

```
void putpixel(int x, int y, int culoare);
```

unde:
`(x,y)` - defineşte poziția punctului;
`culoare` - defineşte culoarea punctului şi este un întreg din intervalul [0,15].

Pentru trasarea liniilor se pot folosi trei funcții: **line, lineto, linerel**. Funcția **line** are prototipul:

```
void line (int xstart,int ystart,int xfin,int yfin);
```

Funcția trasează un segment de dreaptă ale cărui capete sunt punctele de coordonate: `(xstart,ystart)` şi `(xfin,yfin)`.

Funcția **lineto** are prototipul:

```
void   lineto (int x, int y);
```

Ea trasează un segment de dreaptă care are ca origine poziția curentă, iar ca şi punct final cel de coordonate `(x,y)`. Punctul final devine

poziția curentă. De notat că funcția **moveto** permite definirea poziției curente.

Funcția **linerel** are prototipul:

```
void   linerel (int x, int y);
```

Dacă notăm cu `xcrt` şi `ycrt` coordonatele poziției curente, atunci funcția **linerel** trasează un segment de dreaptă ale cărui capete sunt punctele de coordonate: `(xcrt,ycrt)` şi `(xcrt+x,ycrt+y)`.

Alte funcții care permit trasări de figuri geometrice utilizate frecvent sunt:

```
void arc (int xcentru, int ycentru, int
unghistart, int unghifin, int raza);
```

care trasează un arc de cerc; unghiurile sunt exprimate în grade sexagesimale.

```
void circle (int xcentru, int ycentru, int raza);
```

trasează un cerc; `(xcentru,ycentru)` fiind coordonatele centrului arcului de cerc şi respectiv cercului trasat de aceste funcții; parametrul `raza` defineşte mărimea razei curbelor respective.

```
void ellipse (int xcentru, int ycentru, int
unghistart, int unghifin, int semiaxamare, int
semiaxamica);
```

trasează un arc de elipsă cu centrul în punctul de coordonate (xcentru,ycentru) având semiaxa mică definită de parametrul `semiaxamica`, iar semiaxa mare de parametrul `semiaxamare`.

```
void rectangle (int st, int sus, int dr, int jos);
```

trasează un dreptunghi definit de colțurile sale opuse: `(st,sus)` – colțul din stânga sus şi `(dr,jos)` – colțul din dreapta jos.

```
void drawpoly (int nr, int   *tabpct);
```

trasează o linie poligonală, unde:

`nr` - numărul laturilor;

`tabpct` - este un pointer spre întregi care definesc coordonatele liniei poligonale. Acestea sunt păstrate sub forma: `abscisa_i`, `ordonata_i` unde `i` are valorile `1,2,...nr+1`.

Linia poligonală este închisă dacă primul punct coincide cu ultimul (au aceleaşi coordonate). Coordonatele utilizate ca parametri la apelul acestor funcții sunt relative la fereastra activă. Culoarea de trasare este cea curentă.

La trasarea liniilor cu ajutorul funcțiilor: **line, lineto şi linerel** se poate defini un stil de trasare folosind funcția **setlinestyle**. Această funcție are prototipul:

```
void setlinestyle (int stil, unsigned sablon, int grosime);
```

unde:

`stil` - este întreg din intervalul [0,4] care defineşte stilul liniei conform tabelei de mai jos:

Constantă simbolică	***Valoare***	***Stil***
SOLID_LINE	0	Linie continuă
DOTTED_LINE	1	Linie punctată
CENTER_LINE	2	Linie întreruptă formată din liniuțe de două dimensiuni
DASHED_LINE	3	Linie întreruptă formată din liniuțe de aceeaşi dimensiune
USERBIT_LINE	4	Stil definit de utilizator prin şablon

`sablon` - defineşte stilul liniei. Are sens numai când primul parametru –`stil` – are valoarea 4. În rest este neglijat şi de aceea poate avea valoarea zero.

`grosime` - defineşte lăţimea liniei în pixeli. Pot fi două lăţimi: `NORM_WIDTH` (valoarea 1 pixel) şi `THICK_WIDTH` (valoarea 3 pixeli).

Exemplu simplu de verificare a iniţializării modului grafic:

```
#include <graphics.h>

  int main()
  {
      initwindow(400,300); //initializeaza o
ferestra grafica de 400x300 pixeli
      setbkcolor(1);
      cleardevice( );//stabileste culoarea
ecranului (blue).
      moveto(0,0); //plaseaza cursorul în origine
      lineto(50,50); //traseaza o linie pana la
punctul (50,50)
      setcolor(17);
      putpixel(100,130,7);
      line(20,210,400,400);
      while(!kbhit());// asteapta apasarea unei
taste
      closegraph();   // inchide fereastra grafica
      return 0;
  }
```

2.2.5. Ecran virtual si ecran grafic

De cele mai multe ori, pentru o bună reprezentare grafică a obiectelor geometrice, este necesară o conversie a coordonatelor matematice în coordonate de pixeli pe fereastra grafică.

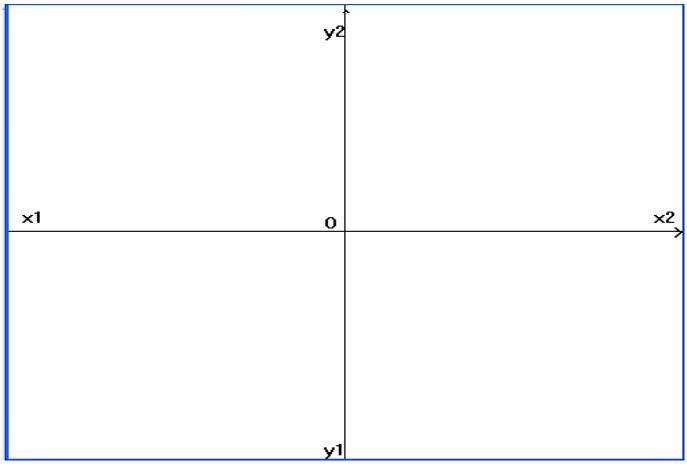


Fig.2.5 Ecranul matematic virtual

Astfel, în ipoteza că avem de reprezentat grafic o figură inclusă în perimetrul definit]n Figura 2.5: [x1,x2]×[y1,y2], se vor folosi două funcții de conversie a coordonatelor matematice în coordonatele naturale pentru C, descrise în Figura 2.4. Pentru aceasta am introdus două funcții speciale de conversie:

$$\begin{cases} x \to xe(x) = \dfrac{x - x1}{x2 - x1} \cdot a \\ y \to ye(x) = -\dfrac{y - y1}{y2 - y1} \cdot b \end{cases}$$

unde *a* şi *b* reprezintă numărul de coloane, respectiv de linii, ale ecranului grafic:

```
a=getmaxx();  b=getmaxy();
```

În continuare am prezentat o secvență de program care implementează aceste conversii şi desenează şi axele matematice ale ecranului virtual.

```
#include <stdio.h>
#include <graphics.h>

float x1,x2,yy1,yy2;
int a,b;

int xe(float x)
```

```
 // normalizarea coocdonatei x
{return((int) floor((x-x1)/(x2-x1)*a));}

int ye(float y)
 // normalizarea coocdonatei y
{return((int) floor((y2-y)/(y2-y1)*b));}

void axe()
{setcolor(0);
  outtextxy(xe(x2)-20,ye(0)-20,"x");
  outtextxy(xe(x2)-6,ye(0)-7,">");
  outtextxy(xe(0)-15,ye(y2)+15,"y");
  outtextxy(xe(0)-15,ye(0)-15,"O");
  outtextxy(xe(0)-1,ye(y2)+1,"^");
 line(xe(x1),ye(0),xe(x2),ye(0));
 line(xe(0),ye(y1),xe(0),ye(y2));
}

int main()
{
 printf("Limitele domeniului orizontal:\n");
 printf("x1="); scanf("%f",&x1);
 printf("x2="); scanf("%f",&x2);
 printf("Limitele domeniului vertical:\n");
 printf("y1="); scanf("%f",&yy1);
 printf("y2="); scanf("%f",&yy2);

 initwindow(800,600, "AXE",200,200);
 setbkcolor(15);
 cleardevice();
 a=getmaxx(); b=getmaxy();
 axe();

 getchar(); getchar();
 closegraph();
 return 0;
}
```

2.3 Teme Laborator/Seminar

1. Desenați o elipsă centrată în mijlocul ferestrei grafice. Plasați 50 de puncte roşii de coordonate arbitrare în interiorul acesteia. (**Indicație**: a se folosi funcția *rand*() din C)
2. Generați 8 triunghiuri aleatoare pe ecranul grafic, schimbând culorile de desenare ale acestora

Capitolul III. Noțiuni geometrice elementare

Descriere Generală

Există un număr vast de sisteme geometrice care pot fi folosite pentru reprezentări grafice şi calcule în geometria computațională: geometria vectorială, geometria afină, geometria Euclidiană şi geometria proiectivă, de exemplu. Cursul 3 utilizează aproape exclusiv geometria afină şi Euclidiană. Înainte să trecem la geometria Euclidiană mai întâi vom aminti câteva noțiuni fundamentale din algebra vectorială şi vom defini oarecum geometria de bază, numită *geometrie afină*. Apoi vom adăuga o operație, numită produs intern, care extinde geometria afină la geometria Euclidiană, definind noțiunea de perpendicularitate, ceea ce permite caracterizarea dreptelor, vectorilor şi planurilor în termeni analitici. Cursul 4 se axează pe definirea principalelor transformări geometrice în plan, precum şi în spațiu.

Obiective

1. Recapitularea cunoştințelor de geometrie cu coordonate (analitică) din liceu
2. Introducerea transformărilor geometrice de bază pentru controlul reprezentărilor grafice.

3.1 Puncte şi vectori

Noțiunea de punct a fost introdusă de către greci în încă din antichitate, dar fără a i se asocia un sistem formal de poziționare. *Punctele* reprezintă principala noțiune primară a geometriei, definindu-se mai mult intuitiv ca *obiectele geometrice de dimensiuni*

nule. Punctele se caracterizează prin localizare; mai precis, prin *coordonate* într-un sistem de coordonate. Sistemele de coordonate au fost introduse de către Fermat şi Descartes pentru a specifica o localizare precisă în spaţiu, reducând astfel rezolvarea unor probleme de geometrie clasică (abordate, de la greci începând, cu rigla şi compasul) la rezolvarea unor ecuaţii algebrice simple.

Orice **sistem de coordonate** are un punct de referinţă absolut, numit **origine**, în raport cu care se localizează restul punctelor, şi un ansamblu de **axe** de coordonate (drepte orientate sau vectori), care determină direcţiile în care se efectuează măsurătorile de localizare. Sistemul de coordonate asociază fiecărui punct *P* dintr-un spaţiu cu *n* dimensiuni un ansamblu de numere $(x_1,...,x_n)$ caracterizând cantitativ operaţiile de deplasare măsurată din origine până în punctul *P* astfel: plecând din origine, ne deplasăm pe distanţa x_1 în direcţia primei axe, pe distanţa x_2 în direcţia celei de-a doua axe, şi aşa mai departe până la ultima axă. Punctul de sosire trebuie să fie *P*. Ansamblul de numere $(x_1,...,x_n)$ poartă numele de **coordonate carteziene** ale punctului *P*, iar ansamblul de axe utilizat pentru asocierea punct-coordonate se mai numeşte şi **reper.**

În cele ce urmează notăm pe scurt $P=(x_i)_{i=1,n}=(x_i)$ şi identificăm originea prin coordonatele $O=(0,0,...,0)$, iar axele prin proprietatea: axa *i* este ansamblul de puncte având toate coordonatele nule, cu excepţia x_i.

Geometria vectorială operează cu două noţiuni fundamentale: *scalarii* şi *vectorii*. **Scalarii** reprezintă simple mărimi de magnitudine, exprimate prin numere aparţinând aceleiaşi mulţimi ca şi coordonatele (în general □). **Vectorii** reprezintă mărimi numerice asociate unei direcţii orientate în spaţiu, şi sunt definiţi printr-un n-uplu $\mathbf{v}= (v_1,...,v_n)$, unde v_i sunt scalari. Interpretarea geometrică a unui vector îl identifică pe acesta cu segmentul orientat de dreaptă care pleacă din origine şi ajunge în punctul $P= (v_1,...,v_n)$, dar, cu toate că au aceeaşi descriere numerică, vectorii şi punctele sunt noţiuni distincte. Mulţimea vectorilor o vom desemna prin □.

Considerăm acum scalarii arbitrari a, b$\in\mathfrak{R}$ şi vectorii $\mathbf{v}=(v_1,...,v_n)$, $\mathbf{w}=(w_1,...,w_n)$. Principalele operaţii care se definesc pe mulţimea vectorilor sunt

a. **Înmulţirea cu scalari**:

„·”:$\mathfrak{R}x\mathfrak{R}\to\mathfrak{R}$, $a\cdot\mathbf{v}=(av_1,...,av_n)$

b. **Adunarea vectorilor**:

„+”:$\mathfrak{R}x\mathfrak{R}\to\mathfrak{R}$, $\mathbf{v+w}=(v_1+w_1,...,v_n+w_n)$.

3.2 Geometrie afină şi euclidiană

3.2.1 Operaţii afine cu puncte, vectori şi scalari

Geometria afină presupune existenţa unei mulţimi de *scalari* (numere reale) □, unei mulţimi de *puncte* $□_n$ (*n* fiind dimensiunea spaţiului în care lucrăm) şi unei mulţimi de *vectori liberi* (sau, mai simplu, *vectori*). Punctele sunt folosite pentru a specifica poziţia vectorilor. Vectorii liberi sunt folosiţi pentru a indica o direcţie şi o mărime, dar nu au o poziţie fixă în spaţiu. Aceasta este principala diferenţă faţă de algebra liniară unde nu există o separare reală între puncte şi vectori.

Operaţiile cu vectori sunt cele familiare din algebra liniară. Este posibil să scădem 2 puncte. Din diferenţa a 2 puncte $p - q$ rezultă un vector liber de la q la p.

Este de asemenea posibil să adunăm un punct la un vector. Din adunarea punct-vector $p + v$ rezultă punctul care este mutat de v din p. Notăm cu S un scalar generic, cu v un vector generic şi cu P un punct generic, operaţiile următoare pot avea loc în geometria afină:

- $S \cdot V \rightarrow V$ multiplicarea vector-scalar
- $V + V \rightarrow V$ adunarea vectorilor
- $P - p \rightarrow V$ scăderea punctelor
- $P + V \rightarrow P$ adunarea punct-vector

Din aceste operaţii pot deriva alte operaţii. De exemplu, putem defini scăderea a 2 vectori $u - v$ ca $\boldsymbol{u} + (-1) \cdot \mathbf{v}$ sau împărţirea scalar-vector v / α ca $(1/\alpha) \cdot \boldsymbol{v}$, dacă $\alpha \neq 0$. Există un vector special, numit *vectorul nul*, 0, care nu are mărime, astfel încat $v + 0 = v$.

Se observă că nu este posibil să înmulţim un punct cu un scalar sau să adunăm 2 puncte. Totuşi există o operaţie specială care combină aceste 2 elemente, numit *combinaţie afină.*

Se dau 2 puncte p_0 şi p_1 şi 2 scalari α_0 şi α_1, astfel încat $\alpha_0 + \alpha_1=1$, definim combinaţia afină:

$$\textbf{\textit{aff}}(p_0,p_1; \alpha_0,\alpha_1)= \alpha_0 p_0 + \alpha_1 p_1 = p_0 + \alpha_1(p_1+p_0) .$$

Termenul din mijlocul ecuaţiei de mai sus este în mod evident incorect fiind dată lista noastră de operaţii. Dar acesta este modul în care operaţia afină este expimată de obicei, numită *media ponderată a 2 puncte*. Termenul din partea dreaptă (care joacă rol de echivalent algebric) este corect. O observaţie importantă este că dacă $p_0 \neq p_1$, atunci punctul *aff*$(p_0,p_1; \alpha_0,\alpha_1)$ se află pe dreapta care uneşte p_0 şi p_1. Deoarece α variază de la $-\infty$ până la $+\infty$, funcţia *aff* uneşte toate punctele de pe dreapta definită de p_0 şi p_1.

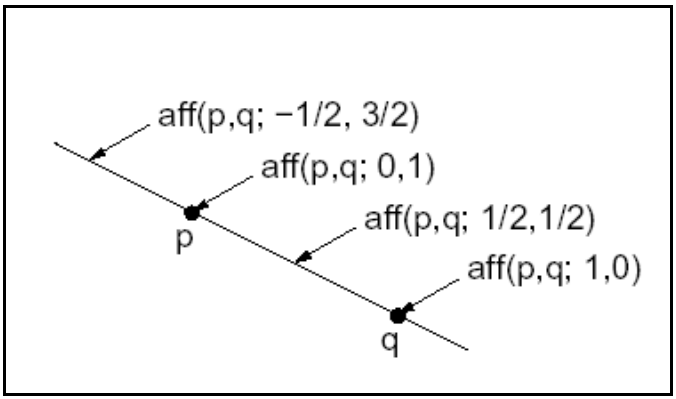


Figura 3.1 Combinații afine ale punctelor p şi q

În cazul particular în care $0 \le \alpha_0,\alpha_1 \le 1$, *aff*($p_0,p_1;\alpha_0,\alpha_1$) este un punct care subdivide segmentul de dreaptă [p_0,$p1$] în 2 subsegmente de mărimi relative la α_0 şi α_1. Operația rezultantă este numită ***combinație convexă*** şi mulțimea tuturor combinațiilor convexe formează segmentul de dreaptă [p_0,$p1$].

Sunt uşor de extins ambele tipuri de combinații la trei puncte, de exemplu, adăugând condiția ca suma $\alpha_0+\alpha_1+\alpha_2=1$.

aff(p_0,p_1,p_2; $\alpha_0,\alpha_1,\alpha_2$) = $\alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 = p_0 + \alpha_1(p_1-p_0) + \alpha_2(p_2-p_0)$.

Mulțimea tuturor combinațiilor afine a trei puncte generează un plan. Mulțimea tuturor combinațiilor convexe de trei puncte generează toate punctele triunghiului definit de puncte. Aceste forme sunt numite *acoperire afină*, sau *închidere afină*, şi respectiv *acoperire convexă* a punctelor.

3.2.2. Produsul scalar

În geometria afină nu am oferit nici o cale de a discuta despre unghiuri sau distanțe. Geometria Euclidiană este o extensie a geometriei afine care cuprinde o operație în plus numită *produs intern*, care reprezintă doi vectori reali (nu puncte) într-un real nenegativ. Un exemplu important de produs intern este *produsul punct*, definit cum urmează. Să presupunem că vectorul *d*-dimensional ***u*** este reprezentat de vectorul de ordonate (neomogene) (u_1, u_2, … , u_d). Apoi definim

$$\langle \mathbf{u}, \mathbf{v} \rangle = \Sigma\, u_i v_I \qquad i=1..d-1$$

Produsul scalar este util la calculul următoarelor entități.

Lungimea: unui vector **v** este definită prin $|\mathbf{v}| = \langle \mathbf{v},\mathbf{v} \rangle$.

Normalizare: Se dă orice vector **v** nenul , definim *normalizarea vectorului* **v** ca un vector de lungimea unei unități care indică aceeaşi direcție ca vectorul **v**. Vom nota aceasta cu v_n:

$$v_n = v/|v|.$$

Distanța dintre puncte: Notăm fie ***dist***(p, q) fie (q,p) lungimea vectorului dintre ele, $|p - q|$.

Unghiul: Dintre doi vectori nenuli u şi v (variind de la 0 la 2π) este

$$\text{ang}(u, v) = \arccos\langle u_n, v_n \rangle = \arccos\langle u,v\rangle/(|\mathbf{u}|\cdot|\mathbf{v}|)..$$

3.3 Drepte şi cercuri în plan

Dreapta este o noțiune primară în geometria euclidiană, în sensul că nu se defineşte geometric, la fel ca şi punctul. Pe de altă parte, din punct de vedere analitic, o dreaptă este definită ca mulțimea punctelor din plan $P(x_P,z_P)$ ale căror coordonate verifică o ecuație liniară:

$$Ax_P+By_P+C=0$$

sau, în forma explicită

$$y_P=mx_P+n$$

unde $m=-B/A$ defineşte *panta* dreptei, caracterizând direcția ei, iar n particularizează drepta printre toate dreptele av\nd acceaşi direcție.

În mod evident o dreaptă poate fi definită în mod unic de un punct fix $M(x_1, y_1)$ şi de un vector liber de direcţie **v**(p,q). Ecuaţia dreptei se scrie astfel

$$(X-x_1)/p = (Y-y_1)/q .$$

Dacă definim dreapta prin două puncte $M(x_1, y_1)$ şi $N(x_2, y_2)$, ecuaţia ei se scrie

$$(X-x_1)/(x_2-x_1) = (Y-y_1)/ (y_2-y_1) .$$

Două drepte sunt paralele dacă au aceeaşi pantă $m=(y_2-y_1)/(x_2-x_1)$. Dacă notăm cu m_1 panta primei drepte şi cu m_2 panta celei de-a doua, condiţia de paralelism se scrie $m_1=m_2$ (cei doi coeficienţi pot fi chiar infoniţi pentru dreptele verticale). Cu aceleaşi notaţii, condiţia de perpendicularitate a două drepte se scrie $m_1m_2=-1$ şi derivă din nulitatea produsului scalar a oricare dou vectori de direcţie corespunzători celor două drepte. Astfel, dacă dreptele sunt definite de puntele $A(x_A,y_A)$ şi $B(x_B,y_B)$, respectiv $C(x_C,y_C)$ şi $A(x_D,y_D)$, condiţia de paralelism se scrie

$$(x_A- x_B)(y_C-y_D) = (x_C- x_D)(y_A-y_B),$$

iar cea de perpendicularitate:

$$(x_A- x_B)(x_C-x_D) + (y_A- y_B)(y_C-y_D) = 0.$$

Cercul de centru $O(x_0,y_0)$ şi rază *r* este definit ca mulţimea punctelor M(x,y) din plan aflate la distanţă fixă *r* de punctul O. Ecuaţia (implicită) a cercului derivă din formula distanţei dintre două puncte şi este:

$$(x-x_0)^2+(y-y_0)^2 = r^2$$

Cercul admite şi o reprezentare prin ecuaţii standard parametrice (care asociază fiecărui punct al cercului un *parametru* real *t* şi doar unul):

$$x= x_0+r\cdot\cos(t),\ \ y= y_0+r\cdot\sin(t),\ t\in[0,2\pi).$$

3.4 Reprezentarea spațiului 3D

3.4.1 Sistemul obiect şi observator

Pentru reprezentarea unui punct în spațiu cu 3D se va alege un anumit sistem de coordonate. Sistemul direct : mâna dreaptă arată cu degetul arătător axa O*x*, cu degetul mare axa O*z*, iar cu cel mijlociu axa O*y*.

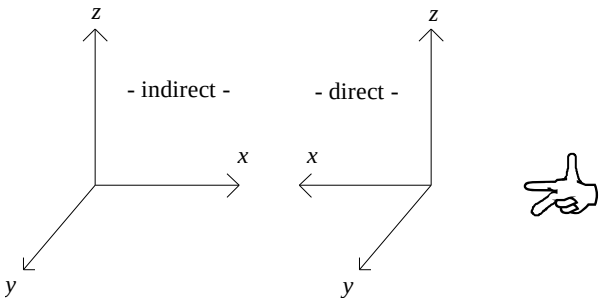


Figura 3.2 Sisteme de coordonatedirecte sau indirecte

Sistemul indirect (sau al mâinii stângi) va avea axa O*x* opusă. Cu alte cuvinte, dacă privim planul *x*O*y* al unui sistem direct, atunci axa Oz este în direcția observatorului. În sistemul indirect, axa Oz este în direcția opusă.

Pentru a repera un punct P al spațiului, decidem să adoptăm sistemul direct, în care vom utiliza trei sisteme de coordonate: *carteziene, sferice* şi *cilindrice*.

Sistemul observator este cel în care originea sistemului de axe coincide cu poziția observatorului. Poate fi la rândul său direct sau indirect.

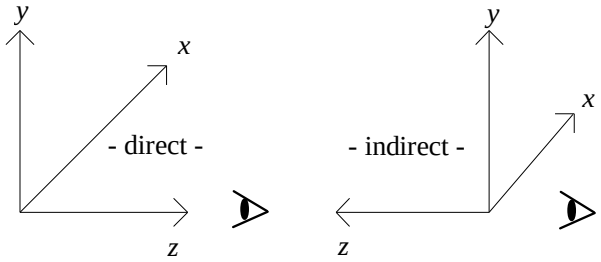
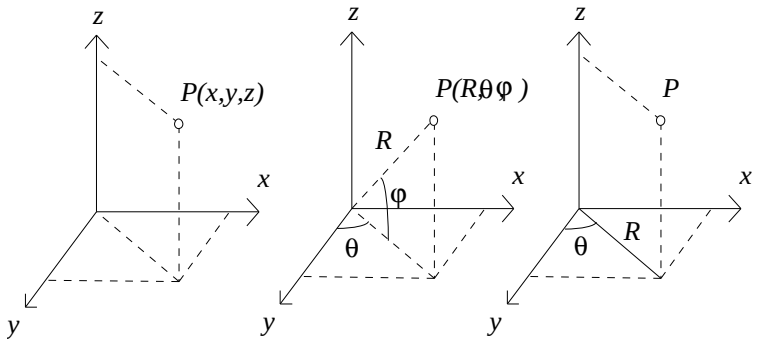


Figura 3.3 Sisteme de observație directe sau indirecte

3.4.2 Sisteme de coordonate 3D

- ***Coordonatele carteziene***. Se precizează: x – proiecția punctului pe axa OX, y – proiecția pe OY şi z – proiecția pe OZ.

- ***Coordonate sferice.*** Se precizează: raza R, unghiului φ (latitudinea) şi a unghiului θ (longitudinea), cu restricțiile:
  $R \in [0, \infty), \theta \in [0, 2\pi], \varphi \in [-\pi/2, \pi/2]$.
  Trecerea de la coordonate carteziene la sferice se face astfel:
  $$R = \sqrt{x^2 + y^2 + z^2},$$
  $$\theta = \begin{cases} \pi/2, & \text{dacă } x = 0,\ y > 0; \\ 3\pi/2, & \text{dacă } x = 0,\ y < 0; \\ \text{arctg}\,(y/x) + k\pi, & \text{dacă } x \neq 0; \end{cases}$$
  unde: $k = 0$ pentru $x > 0$, $y > 0$; $k = 1$ pentru $x < 0$; $k = 2$ pentru $x > 0$, $y < 0$;
  $$\varphi = \begin{cases} -\pi/2, & \text{dacă } x^2 + y^2 = 0,\ z < 0; \\ \pi/2, & \text{dacă } x^2 + y^2 = 0,\ z > 0; \\ \text{arctg}\,(z/\sqrt{x^2 + y^2}), & \text{dacă } x^2 + y^2 \neq 0. \end{cases}$$



a) *b)* *c)*

Figura 3.4 Sisteme de coordonate: a) carteziene; b) sferice; c) cilindrice

Formulele de trecere de la coordonate sferice la carteziene sunt: $x = R \cos\theta \cos\varphi$; $y = R \sin\theta \cos\varphi$; $z = R \sin\varphi$.

- ***Coordonatele cilindrice*** ale unui punct *P* sunt: *cota* (*z*), *raza* (*R*) şi *unghiul* (θ) cu: $z \in \mathbf{R}$, $R \in [0, \infty)$, $\theta \in [0, 2\pi]$. Trecerea din coordonate carteziene în cilindrice se realizează prin relaţiile:

$$R = \sqrt{x^2 + y^2}\,;\ z = z;\ \theta = \begin{cases} \pi/2, & \text{dacă } x = 0,\ y > 0; \\ 3\pi/2, & \text{dacă } x = 0,\ y < 0; \\ \arctg(y/x) + k\pi, & \text{dacă } x \neq 0; \end{cases}$$

după aceeaşi relaţie ca la coordonatele sferice, iar trecerea inversă, de la coordonate cilindrice la carteziene, este dată prin: $x = R \cos\theta$; $y = R \sin\theta$; $z = z$.

3.5 Drepte, planuri şi sfere în spațiu

Planul în spațiu se defineşte de manieră absolut asemănătoare cu dreapta în plan. Un plan este definit ca mulțimea punctelor din spațiu P(x,y,z) ale căror coordonate verifică o ecuație liniară:

$$Ax+By+Cz+D=0$$

sau, în forma explicită:

$$y=mx+ny+p.$$

Dreapta în spațiu este definită întotdeauna ca intersectie a două plane:

$$A_1x+B_1y+C_1z+D_1=0$$

$$A_2x+B_2y+C_2z+D_2=0$$

Pe de altă parte, fiind date un punct fix $M(x_1, y_1, z_1)$ şi de un vector liber de direcție **v**(p,q,r), ecuația dreptei determinată de aceştia este

$$(X-x_1)/p = (Y-y_1)/q = (Z-y_1)/r.$$

Dacă definim dreapta prin două puncte $M(x_1, y_1, z_1)$ şi $N(x_2, y_2 z_2)$, ecuația ei se scrie

$$(X-x_1)/(x_2-x_1) = (Y-y_1)/ (y_2-y_1) = (Z-z_1)/ (z_2-z_1).$$

Două drepte sunt paralele dacă au vectorii de direcție $v=($ (x_2-x_1), $(y_2-y_1))$ proporționali. Astfel, dacă dreptele sunt definite de puntele $A(x_A,y_A)$ şi $B(x_B,y_B)$, respectiv $C(x_C,y_C)$ şi $A(x_D,y_D)$, condiția de paralelism se scrie

$$(x_A- x_B)(y_C-y_D) = (x_C- x_D)(y_A-y_B) ,$$

$$(x_A- x_B)(z_C-z_D) = (x_C- x_D)(z_A-z_B) ,$$

Condiția de perpendicularitate derivă din anularea produsului scalar a celor doi vectori de direcție:

$$(x_A- x_B)(x_C-x_D) + (y_A- y_B)(y_C-y_D) + (z_A- z_B)(z_C-z_D) = 0.$$

Sfera de centru $O(x_0,y_0,z_0)$ şi rază *r* este definită ca mulțimea punctelor M(x,y,z) din spațiu aflate la distanță fixă *r* de punctul O. Ecuația (implicită) a sferei derivă din formula distanței dintre două puncte în spațiu şi este:

$$(x-x_0)^2+(y-y_0)^2 +(y-y_0)^2 = r^2$$

Ecuațiile standard parametrice ale sferei asociază fiecărui punct doi *parametrii* reali *t* şi *s*):

$$x= x_0+r\cdot cos(t)cos(s),\ y= y_0+r\cdot sin(t)cos(s),\ z= z_0+r\cdot sin(s)$$

$$t\in[0,2\pi),\ s\in[0,2\pi).$$

3.6. Reprezentarea spațiului 3D pe ecran

Principala problemă care se pune când vorbim despre obiecte tridimensionale în grafica pe calculator este de a reprezenta pe ecranul bi-dimensional puncte având trei coordonate spațiale. În acest caz, se alege ca axă verticală axa *Oz* în loc de *Oy*, înlocuind funcția de renormare a coordonatelor verticale `int ye(float y)` din capitolul precedent cu

```
int ze(float z)
 // normalizarea coocdonatei y
{return((int) floor((2*x2-y)/(2*x2-2*x1)*yemax));}
```

unde dimensiunea pe verticală a ferestrei de vedere este stabilită la aceeaşi mărime cu cea orizontală: (x_1,x_2).

A treia axă, *Oy*, se introduce ca o semidreaptă ce plecă din origine spre stănga-jos, cu un unghi arbitrar fixat *alfa*.

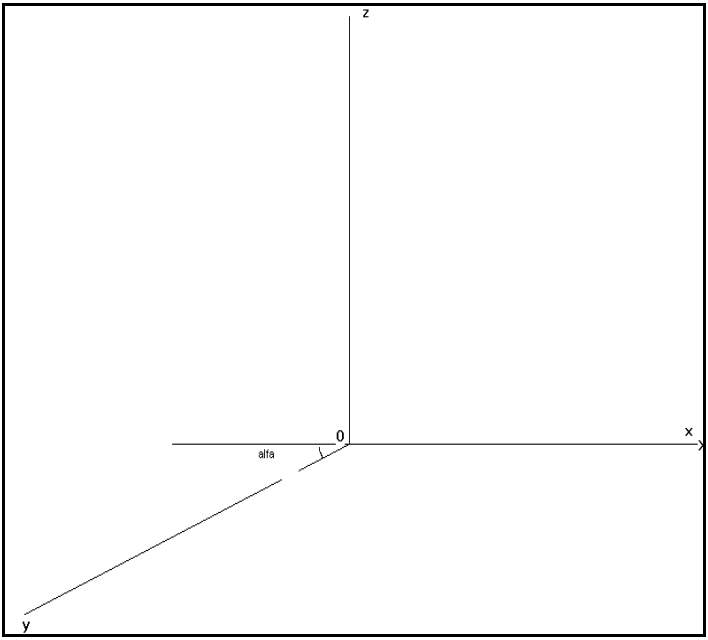


Fig.3.5 Axele în reprezentarea 3D

Transformarea de coordonate de la 3D la 2D se obține prin proiecție:

$$(x,z,y) \rightarrow (x\text{-}\cos(alfa)*y,\ z\text{-}\sin(alfa)*y) \rightarrow$$

$$(\mathbf{xe}(x\text{-}\cos(alfa)*y),\ \mathbf{ze}(z\text{-}\sin(alfa)*y))$$

Am prezentat în paginile următoare un exemplu de program care reprezintă grafic o elice conică 3D (o «bobină» înfăşurată pe un con).

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <math.h>
#define pi 3.14;

float x1,x2,yy1,yy2, alfa=3.1415/8;
```

```
int xemax,yemax;

int xe(float x)
 // normalizarea coocdonatei x
{return((int) floor((x-x1)/(x2-x1)*xemax));}

int ze(float z)
 // normalizarea coocdonatei y
{return((int) floor((yy2-z)/(yy2-yy1)*yemax));}

void axe()
{setcolor(0);
 line(xe(x1/2),ze(0),xe(x2),ze(0));
 line(xe(0),ze(0),xe(0),ze(yy2));

line(xe(0),ze(0),xe(0+x1*cos(alfa)),ze(0+x1*sin(al
fa)));
 outtextxy(xe(0)-15,ze(0)-15,"O");
 outtextxy(xe(x2)-20,ze(0)-20,"x");
   outtextxy(xe(x2)-6,ze(0)-7,">");
 outtextxy(xe(0)+15,ze(yy2)+5,"z");
  outtextxy(xe(0)-1,ze(yy2)+1,"^");
outtextxy(xe(x1*cos(alfa)),ze(x1*sin(alfa)),"y");
}

float f(float theta){
 float pie=3.14;
 return (2*0.7*sin(theta)*tan(theta));
}


void grafic() {
 float t,h;
 float x,y,z;
 float pie=3.1415;
 t=0; h=pie/800;
 while (t<=+15)
    {x=0.2*t*cos(3*t);  //ecuatiile elicei conice
    y=0.2*t*sin(3*t);
    z=0.4*t;
    putpixel(xe(x-y*cos(alfa) ),ze(z-
y*sin(alfa)),3);
    t=t+h;
    }
 t=0;
}
```

```
int main()
{int gd,gm;

 printf("Limitele domeniului orizontal:\n");
 printf("x1="); scanf("%f",&x1);
 printf("x2="); scanf("%f",&x2);
 yy1=x1; yy2=x2;

 initwindow(800,800, "Elice conica",200,200);
 setbkcolor(15);
 cleardevice();
 xemax=getmaxx(); yemax=getmaxy();
 axe();
 setcolor(2);
 grafic();

 getchar(); getchar();
 closegraph();
 return 0;}
```

3.7. Teme de laborator

Date :

- Pentru reprezentarea vectorilor în plan definim tipul:

```
typedef struct{
      float x,y;
}vector;
```

- Produsul scalar a doi vectori se calculează cu ajutorul funcţiei:

```
float produs_scalar(vector u,vector v){
      return (u.x*v.x+u.y*v.y);
      }
```

Cosinusul unghiului dintre doi vectori se calculează folosind funcţia:

```
float cos_unghi(vector u, vector v){
     return
(produs_scalar(u,v)/(sqrt((produs_scalar(u,u
))*(produs_scalar(v,v)))));
     }
```

- Desenați un triunghi pe ecran şi determinații centrul de greutate. Desenați medianele sale.
- Includeți bisectoarele unghiurilor.

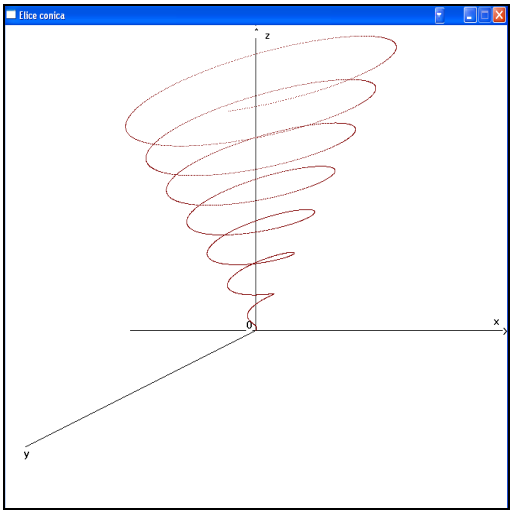


Fig.5.2 Elice conică în reprezentarea 3D

Capitolul IV. Transformări geometrice

Descriere Generală

Capitolul se axează pe definirea principalelor transformări geometrice în plan, precum şi în spațiu. Am introdus de asemenea modelul camerei de luat vederi.

Obiective

Introducerea transformărilor geometrice de bază pentru controlul reprezentărilor grafice.

4.1 Transformări geometrice elementare în plan

Dacă x şi y sunt coordonatele unui punct M, ecuațiile:

$$x' = f(x, y), \quad y' = g(x, y)$$

permit trecerea de la punctul $M(x, y)$ la punctul $M'(x', y')$. Aceste ecuații definesc o transformare plană. Punctul M' este transformatul lui M.

Putem considera coordonatele unui punct ca o matrice 2×1 (un vector) $^{\mathbf{T}}\mathbf{(x,y)}$ (^{T} semnifică "transpus"). Fie produsul matriceal următor

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = (Ax + Cy \quad Bx + Dy).$$

Toate punctele planului xOy înmulțite cu o matrice 2×2 vor da noi puncte (x', y') conform acestor relații.

Există însă tipuri de transformări care nu pot fi reprezentate printr-o matrice 2×2.

4.1.1 Translația

Translația reprezintă deplasarea unei figuri geometrice paralel cu un vector dat **v=(a,b)**.Ea este reprezentată de ecuațiile:

$$x' = x + a, \quad y' = y + b.$$

Familia de transformări depinde de parametrii *a* şi *b*. Dacă $a = b = 0$, atunci punctul $M(x, y)$ rămâne neschimbat şi obținem transformarea identică. Familia conține şi transformarea inversă: $x = x' - a$, $y = y' - b$.

Se observă că nici un produs al matricei generale c nu ne permite să găsim relațiile de translație. Pentru rezolvarea situației se va introduce o a treia componentă a vectorilor (x, y) şi $(x' y')$, obținând $(x\ y\ 1)$, $(x'\ y'\ 1)$. Matricea transformrilor va fi obligatoriu de dimensiune 3x3.
Astfel:

$$\begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}.$$

Spunem că reprezentarea poziției unui punct în plan printr-un vector cu trei componente este o reprezentare în ***coordonate omogene***.

Matricele de dimensiuni 3×3 dau posibilitatea efectuării transformărilor inverse dând familiilor de transformări caracteristica de grup algebric.

4.1.2 Scalarea

Scalarea cu factorii S_x, S_y pe axele OX, OY se defineşte ca transformarea care modifică o imagine prin redimensionarea ei cu factorul de multiplicitate S_x pe orizontală şi S_y pe verticală. Ecuațiile carteziene ale scalării sunt

$$\begin{cases} X^* = S_x \cdot X \\ Y^* = S_y \cdot Y \end{cases}$$

şi are matricea de transformare

$$S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4.1.3 Simetria

Simetriile sunt cazuri particulare de scalări cu factori unitari negativi. Există două tipuri de simetrii: față de o dreaptă sau față de un punct. Cele mai simple simetrii sunt cele față de axe sau origine.

- Simetria în raport cu axa OX:

$$S^X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

- Simetria în raport cu axa OY:

$$S^Y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

- Simetria în raport cu originea O:

$$S^O = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

4.1.4 Rotația

Este considerată transformare elementară rotația în jurul originii, în sens trigonometric, cu un un unghi oarecare, θ.

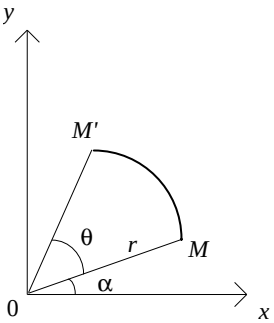


Figura 4.1 Rotația de centru O şi unghi θ.

$$x = r \cdot \cos(\alpha)$$

$$x^t = r \cdot \cos(\theta + \alpha) = r(\cos\theta\cos\alpha - \sin\theta\sin\alpha) = x\cos\theta - y\sin\theta$$

$$y = r \cdot \sin(\alpha)$$

$$y^t = r \cdot \sin(\theta + \alpha) = r(\sin\theta\cos\alpha + \cos\theta\sin\alpha) = x\sin\theta + y\cos\theta$$

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}.$$

4.2 Transformări geometrice elementare în spațiu

Exprimarea transformărilor geometrice tridimensionale într-o formă matricială a fost realizată folosind **coordonatele omogene 3D**. Prin definiție, *coordonatele omogene* ale unui punct de *coordonate carteziene* (X, Y, Z) sînt (kX, kY, kZ, k) unde k este o constantă arbitrară, nenulă. Punctul

$$w = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$ va avea în coordonate omogene vectorul $$w_h = \begin{bmatrix} kX \\ kY \\ kZ \\ k \end{bmatrix}$$

Se observă faptul ca trecerea de la coordonate omogene la coordonate carteziene se face împărțind primele trei valori la cea de a patra.

4.2.1 Translația

Dorim să translatăm un punct de coordonate (X,Y,Z) la o nouă poziție definită prin deplasamentul (X_0,Y_0,Z_0) Transformarea se face utilizând următoarele ecuații

$$X^* = X + X_0$$
$$Y^* = Y + Y_0$$
$$Z^* = Z + Z_0$$

Acest sistem de ecuații se poate scrie în formă matricială astfel

$$\begin{bmatrix} X^* \\ Y^* \\ Z^* \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Pentru a putea concatena mai multe transformări este util să folosim matrici pătrate. Ecuația de mai sus se poate scrie şi astfel

$$\begin{bmatrix} X^* \\ Y^* \\ Z^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

sau

$$v^* = Tv$$

unde v este vectorul coloană al punctului original, v^* vectorul coloană al punctului transformat iar T matricea transformării.

4.2.2 Scalarea

Scalarea cu factorii S_x, S_y, S_z pe axele X, Y, Z se defineşte ca

$$X^* = S_x \cdot X$$
$$Y^* = S_y \cdot Y$$
$$Z^* = S_z \cdot Z$$

şi are matricea de transformare

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.2.3 Rotația

Cea mai simplă formă de rotație este cea a unui punct în jurul uneia dintre axele de coordonate. Rotirea unui punct în jurul unui punct arbitrar implică trei transformări. Prima translatează punctul arbitrar în origine, a doua face rotația şi a treia translatează punctul în poziția inițială.

Rotirea unui punct în raport cu axa Z cu unghiul θ are matricea de transformare

$$R_\theta = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotirea unui punct în raport cu axa X cu unghiul α are matricea de transformare

$$R_\alpha = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotirea unui punct în raport cu axa Y cu unghiul β are matricea de transformare

$$R_\beta = \begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.2.4 Concatenarea şi transformarea inversă

Aplicarea mai multor transformări succesive se poate reprezenta printr-o singură matrice de transformare. De exemplu: translația, scalarea şi rotația în jurul axei Z a unui punct v este dată de

$v^* = R_\theta(S(Tv)) = Av$ unde $A = R_\theta ST$

Exemplu.

Rotație în jurul unei axe arbitrare. Se ştie că rezultatul compunerii unei serii de transformări se poate reprezenta printr-o matrice unică. Ne propunem să căutăm matricea de rotație sub un unghi θ în jurul dreptei *PQ*, paralelă cu O*x*, dată de *P*(*a*, *b*, *c*), *Q*(*d*, *b*, *c*). Metoda determinării matricii transformării se descrie pe scurt astfel: 1) se efectuează o translație *T*, care aduce punctul *P* în originea sistemului de axe; 2) se efectuează rotația *R* cerută de unghiul θ; 3) se execută translația *T'* opusă primei, care readuce punctul *P* la poziția sa inițială.

Se obține produsul matriceal $M = T \cdot R \cdot T'$:

$$M = \begin{Vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & -b & -c & 1 \end{Vmatrix} \cdot \begin{Vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{Vmatrix} \cdot \begin{Vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{Vmatrix} =$$

$$= \begin{Vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & -b\cos\theta + c\sin\theta + b & -b\sin\theta - c\cos\theta + c & 1 \end{Vmatrix}$$

Deşi pînă acum s-a vorbit despre transformarea unui singur punct, problema se poate extinde la transformarea unui set de m puncte simultan, fie acestea v_1, v_2, …, v_m. Atunci matricea V de dimensiune 4 x m poate fi transformată prin operația $V^* = AV$.

În general, pentru orice transformare se poate defini o transformare inversă. Aceasta se obține aplicând ca matrice de transformare inversa matricii transformării originale. De multe ori matricea transformării inverse se obține prin observații simple de natură geometrică. De exemplu

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -X_0 \\ 0 & 1 & 0 & -Y_0 \\ 0 & 0 & 1 & -Z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_\theta^{-1} = \begin{bmatrix} \cos(-\theta) & \sin(-\theta) & 0 & 0 \\ -\sin(-\theta) & \cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

În alte cazuri matricea inversă se obține prin calcul analitic sau numeric.

4.3. Transformarea de perspectivă

Transformarea de perspectivă este cea care proiectează puncte din spațiul tridimensional într-un plan. Ea modelează, deci, transformarea unei scene reale într-o imagine bidimensională. Spre deosebire de transformările prezentate în paragraful anterior, transformarea de perspectivă este neliniară, deoarece implică operații de împărțire la valori ale coordonatelor

În Fig.3.2. se prezintă un model al procesului de formare a unei imagini. Alegem sistemul de coordonate al camerei de luat vederi (x,y,z) aşa fel încât planul imaginii să coincidă cu planul (x,y) şi

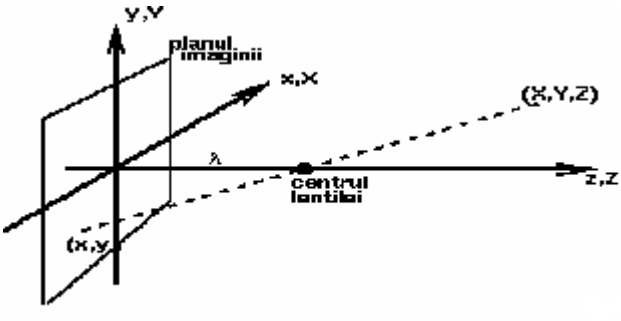


Fig 4.2 Formarea imaginii fotografice

centrul imaginii să fie în originea sistemului. Atunci centrul sistemului optic al camerei este în punctul (0,0,λ) unde λ este distanța focală a lentilei. Presupunem de asemenea că sistemul de coordonate al camerei şi sistemul de coordonate globale (X, Y, Z) coincid.

Fie punctul de coordonate (X, Y, Z) cu Z > λ, deci aflat în fața lentilei. Dorim să aflăm care sînt coordonatele (x, y) ale

proiecției sale în planul imaginii. Din triunghiuri asemenea se obține imediat

$$\frac{x}{\lambda} = -\frac{X}{Z-\lambda} = \frac{X}{\lambda - Z}$$
$$\frac{y}{\lambda} = -\frac{Y}{Z-\lambda} = \frac{Y}{\lambda - Z}$$

unde semnul minus din fața fracțiilor arată că imaginea este inversată, aşa cum rezultă şi din figură. Avem deci

$$x = \frac{\lambda X}{\lambda - Z}; \qquad y = \frac{\lambda Y}{\lambda - Z}$$

Dacă definim transformarea de perspectivă prin matricea

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{-1}{\lambda} & 1 \end{bmatrix}$$

atunci produsul $c_h = Pv_h$ este

$$c_h = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{-1}{\lambda} & 1 \end{bmatrix} \begin{bmatrix} kX \\ kY \\ kZ \\ k \end{bmatrix} = \begin{bmatrix} kX \\ kY \\ kZ \\ -\frac{kZ}{\lambda} + k \end{bmatrix}$$

sau, în coordonate carteziene

$$c = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \dfrac{\lambda X}{\lambda - Z} \\ \dfrac{\lambda Y}{\lambda - Z} \\ \dfrac{\lambda Z}{\lambda - Z} \end{bmatrix}$$

Primele două coordonate sînt chiar coordonatele proiecției punctului w în planul imaginii. Cea de a traia nu are nici o semnificație în termenii modelului din Fig. 3.1. Ea acționează ca o variabilă liberă la transformarea inversă de perspectivă. Matricea transformării inverse de perspectivă este

$$P^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \dfrac{1}{\lambda} & 1 \end{bmatrix}$$

Fie punctul (x_0, y_0) din imagine. Coordonatele sale omogene sînt

$$c_h = \begin{bmatrix} kx_0 \\ ky_0 \\ 0 \\ k \end{bmatrix}$$

Aplicând transformarea inversă rezulta $w_h = P^{-1}c_h$

$$w_h = \begin{bmatrix} kh_0 \\ ky_0 \\ 0 \\ k \end{bmatrix} \qquad si \qquad w = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ 0 \end{bmatrix}$$

Rezultatul Z=0 se datorează faptului că transformarea de perspectivă este o transformare de tip mulți-la-unul. Punctul (x_0, y_0) corespunde unei drepte din spațiul tridimensional care trece prin punctele (x_0, y_0, 0) şi (0, 0, λ). Ecuațiile acestei drepte în spațiul coordonatelor globale sînt

$$X = \frac{x_0}{\lambda}(\lambda - Z)$$

$$Y = \frac{y_0}{\lambda}(\lambda - Z)$$

Ecuațiile arată că fără a cunoaşte ceva despre punctul di spațiul tridimensional care a generat punctul (x_0, y_0) , de exemplu coordonata Z, nu se pot reconstitui coordonatele punctului. Dacă în vectorul punctului c_h introducem variabila liberă z în loc de valoarea 0 obținem

$$c_h = \begin{bmatrix} kx_0 \\ ky_0 \\ z \\ k \end{bmatrix}$$

$$w_h = \begin{bmatrix} kh_0 \\ ky_0 \\ kz \\ \frac{kz}{\lambda}k \end{bmatrix} \qquad si \qquad w = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \frac{\lambda x_0}{\lambda + z} \\ \frac{\lambda y_0}{\lambda + z} \\ \frac{\lambda z}{\lambda + z} \end{bmatrix}$$

Rezolvând sistemul de ecuații care apare prin egalarea celor două exprimări ale vectorului w aşa fel încât să eliminăm variabila liberă z rezultă aceleaşi ecuații deduse mai sus.

$$X = \frac{x_0}{\lambda}(\lambda - Z)$$

$$Y = \frac{y_0}{\lambda}(\lambda - Z)$$

3.7. Modelul camerei de luat vederi

Vom dezvolta în continuare ecuațiile care dau vectorii c_h şi c în situația în care, spre deosebire de paragraful anterior, sistemul de coordonate al camerei nu este aliniat cu cel global.

Conform Fig. 3.3. vom presupune o cameră aflată pe un suport vertical. Camera se poate roti pe suport, în plan orizontal, cu unghiul θ în jurul axei Z. De asemenea, camera se poate înclina cu unghiul α față de axa X. Rotirea o definim ca fiind unghiul dintre axele x şi X, iar înclinarea ca unghiul dintre axele z şi Z.

Punctul a cărui imagine se proiectează în cameră este w, iar proiecția sa este c. Offsetul dintre originea sistemului de coordonate al camerei (centrul imaginii) şi centrul mecanismului de rotire a camerei este dat de vectorul r iar poziția centrului mecanismului în sistemul de coordonate globale este w_0.

Ceea ce dorim să facem este sa aplicăm o serie de transformări elementare, pentru a alinia camera la sistemul de coordonate global, apoi să aplicăm transformarea de perspectivă. Presupunând că, inițial, camera se găsea aliniată cu sistemul de coordonate global, ea ajunge în poziția din fig.3.3. după următoarele mişcări

(1) deplasarea suportului cu vectorul w_0
(2) rotirea în plan orizontal cu unghiul θ
(3) înclinarea cu unghiul α

(4) deplasarea planului imaginii față de centrul mecanismului cu vectorul r.

Deplasarea suportului este o translație dată de matricea

$$G = \begin{bmatrix} 1 & 0 & 0 & -X_0 \\ 0 & 1 & 0 & -Y_0 \\ 0 & 0 & 1 & -Z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

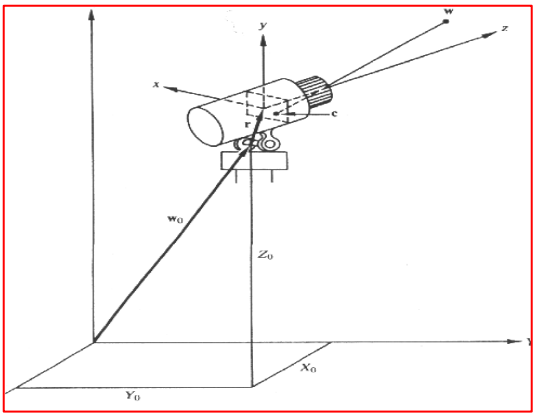


Fig 4.3 Camera de luat vederi

Rotirea şi înclinarea sînt concatenarea a două rotiri elementare matricea de transformare este $R = R_\alpha R_\theta$. Prin calcul rezultă

$$R = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta\cos\alpha & \cos\theta\cos\alpha & \sin\alpha & 0 \\ \sin\theta\sin\alpha & -\cos\theta\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

În fine, deplasamentul cu vectorul r este o translație dată de matricea

$$C = \begin{bmatrix} 1 & 0 & 0 & -r_1 \\ 0 & 1 & 0 & -r_2 \\ 0 & 0 & 1 & -r_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

În acest fel, aplicând asupra punctului w_h seria de transformări $CRGw_h$ coordonatele camerei în coincidență cu cele globale. Acum putem aplica transformarea de perspectivă şi obținem

$$c_h = PCRGw_h$$

Explicitând ecuația matricială de mai sus şi trecând în coordonate carteziene se obțin, ca şi în paragraful anterior, expresiile coordonatelor x, y, ca funcții de coordonatele globale X, Y, Z, cu parametri λ, X_0, Y_0, Z_0, θ, α, r_1, r_2, r_3.

4.4. Teme de laborator

1. Implementați câteva funcții care să modeleze rotația, translația şi scalarea în plan.
2. Desenați un pătrat pe ecran. Încercați să-l rotiți cu un unghi mic (5 grade) de 10 ori, în aceleaşi coordonate, schimbându-i culoarea la fiecare rotație.
3. Translatați segmentul (20,40.260,320) cu vectorul (5,180) pe ecran. (Indicație: folosiți `cleardevice()`pentru a şterge vechea poziție)
4. Translatați un triunghi pe ecran, simulând o animație simplă

Soluție:

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <math.h>
#define pi 3.14;
```

```
float x1,x2,yy1,yy2;
int xemax,yemax;
float pie=3.1415;

int xe(float x)
 // normalizarea coocdonatei x
{return((int) floor((x-x1)/(x2-x1)*xemax));}

int ye(float y)
 // normalizarea coocdonatei y
{return((int)              floor((yy2-y)/(yy2-
yy1)*yemax));}

void axe()
{setcolor(15);
 line(xe(x1),ye(0),xe(x2),ye(0));
 line(xe(0),ye(yy1),xe(0),ye(yy2));
 outtextxy(xe(0)-15,ye(0)-15,"O");
 outtextxy(xe(x2)-20,ye(0)-20,"x");
   outtextxy(xe(x2)-6,ye(0)-7,">");
 outtextxy(xe(0)-15,ye(yy2)+15,"y");
  outtextxy(xe(0)-1,ye(yy2)+1,"^");

}

float f(float theta){
 float pie=3.14;
 return (2*0.7*sin(theta)*tan(theta));
}


void grafic() {
  float a=3,b=2; // vectorul (a,b)
 int i,c;
  int xa,ya,xb,yb,xc,yc;
  xa=7; ya=10;
  xb= 10; yb=4;
```

```
  xc=1; yb=1;
  setbkcolor(0);
  cleardevice();
  for(i=1;i<10; i++)
    {axe();
  c= rand()%14+1;
// deseneaza triunghi
  setcolor(c);
 outtextxy(xe(xa)+10,ye(ya)-10,"A");
 circle(xe(xa),ye(ya),2);
 outtextxy(xe(xb)+10,ye(yb)-10,"B");
 circle(xe(xb),ye(yb),2);
 outtextxy(xe(xc)-10,ye(yc)+10,"C");
 circle(xe(xc),ye(yc),2);

 line(xe(xa),ye(ya),xe(xb),ye(yb));
 line(xe(xc),ye(yc),xe(xb),ye(yb));
 line(xe(xa),ye(ya),xe(xc),ye(yc));

 while(!bkhit());

  delay(700);
  //sterge triunghi
  setcolor(0);
 outtextxy(xe(xa)+10,ye(ya)-10,"A");
 circle(xe(xa),ye(ya),2);
 outtextxy(xe(xb)+10,ye(yb)-10,"B");
 circle(xe(xb),ye(yb),2);
 outtextxy(xe(xc)-10,ye(yc)+10,"C");
 circle(xe(xc),ye(yc),2);

 line(xe(xa),ye(ya),xe(xb),ye(yb));
 line(xe(xc),ye(yc),xe(xb),ye(yb));
 line(xe(xa),ye(ya),xe(xc),ye(yc));

 //efectueaza translatie
  xa=xa+a; ya=ya+b;
  xb=xb+a; yb=yb+b;
```

```
    xc=xc+a; yc=yc+b;
  }



}

int main()
{int gd,gm;

 printf("Limitele domeniului orizontal:\n");
 printf("x1="); scanf("%f",&x1);
 printf("x2="); scanf("%f",&x2);
 printf("Limitele domeniului vertical:\n");
 printf("y1="); scanf("%f",&yy1);
 printf("y2="); scanf("%f",&yy2);
 initwindow(800,600, "Exemplu lab 2",200,200);
 xemax=getmaxx(); yemax=getmaxy();
 axe();
 setcolor(YELLOW);
 grafic();

 getchar(); getchar();
 closegraph();
 return 0;
}
```

5. Pentru programul de mai sus, implementați o rotație cu centrul în origine. Eventual modificați centrul de rotație.

Indicație: rotația se descrie prin

```
xa=xa*cos(t)+ya*sin(t);
ya=-xa*sin(t)+ya*cos(t);
xb=xb*cos(t)+yb*sin(t);
yb=-xb*sin(t)+yb*cos(t);
```

```
xc=xc*cos(t)+yc*sin(t);
yc=-xc*sin(t)+yc*cos(t);
```

unde t este un parametru fix (t=3.1415/36 de exemplu)

Capitolul V. Algoritmi elementari de geometrie computațională

Descriere Generala

Acest capitol îşi propune o introducere uşoară în proiectarea algoritmilor caracteristici geometriei computaționale, prezentând elemente algoritmice necesare înțelegerii capitolelor următoare.

Obiective

- Definirea orientării unui triunghi şi utilizarea acesteia pentru a testa existența intersecției a două segmente.
- Testarea apartenenței unui punct la interiorul sau exteriorul unui triunghi sau a unui poligon.
- Construcția unei linii poligonale simple printr-un număr de puncte date

Capitolul de față se bazează pe partea întâia a cărții lui M. de Berg, M. van Kreveld, M. Overmars, şi O. Schwarzkopf, *Computational geometry, algorithms and applications* [1], pe Capitolul 24 din R. Sedgewick – *Algorithms* [7] şi Capitolul 3 din David M. Mount - *CMSC 754 Computational Geometry* [5].

5.1 Puncte, linii şi poligoane

Cele mai multe dintre programele pe care le vom studia în cadrul algoritmilor geometrici operează cu obiecte geometrice simple definite în spațiul bidimensional (rareori în cel tri-dimensional). Obiectul fundamental este *punctul*, pe care îl vom considera ca fiind o pereche sau un triplet de întregi - coordonatele punctului în sistemul cartezian uzual. *Liniile* sunt definite de o pereche de puncte, care sunt legate intre ele printr-un segment de dreaptă.

Pentru a lucra cu aceste figuri geometrice trebuie sa decidem cum le vom reprezenta. Cele mai multe dintre programele noastre vor utiliza urmatoarele reprezentări :

```
typedef  struct [ int x,y;} punct;
typedef  struct [ point p1,p2;} dreapta;
```

Notăm că punctele sunt definite deocamdată în coordonate întregi, din considerente de reprezentare grafică pe ecran. O reprezentare reală poate fi utilizată de asemenea. Considerând numai punctele cu coordonate întregi, vom opera cu algoritmi mai simpli şi mai eficienți.

Obiectele geometrice mai complicate vor fi reprezentate în termenii acestor componente de bază. De exemplu, *poligonul* este o lista de puncte:

```
punct P[200];
```

unde am considerat implicit că punctele succesive sunt legate între ele prin linii şi primul punct este legat de ultimul pentru a forma o figura închisă.

De asemenea, este utilă pentru unele aplicații includerea unor extra-informații asociate cu fiecare punct sau linie. Aceasta poate fi realizată prin adaugarea unei zone adiționale de informații la definitea punctelor.

```
typedef  struct {char nume;
                 int x,y;} punct;
```

Punctele sunt notate cu litere uzual în geometrie. Programele nu au nici un motiv de a se referi la puncte prin “nume”. Bineînteles, ordinea în care punctele apar în figură poate fi importantă în unele programe: într-adevăr, scopul anumitor algoritmi geometrici este sortarea punctelor într-o ordine particulară. Literele pe care le eventual le utilizăm sunt repartizate în ordinea în care punctele vor aparea la introducetera datelor.

Cu titlu de exemplu, putem considera un şir de 16 puncte aleatoare în plan:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
x:	3	11	6	4	5	8	1	7	9	14	10	16	15	13	2	12
y:	9	1	8	3	15	11	6	4	7	5	13	14	2	16	12	10

(5.1)

Un program de geometrie computationala va stoca un în locul şirului de mai sus un vector **p**[1..N] de puncte, modelate ca N perechi de numere întregi. dacă lucrăm cu un poligon, se adiționeaza doua puncte virtuale **p**[0] = **p**[N] şi **p**[1] = **p**[N+1].

- Funcția următoare realizează citirea unui poligon:

```
void citire_poligon(){
      printf("N=");
      scanf("%d",&N);
      for (i=0;i<n;i++){
          printf("dati      numele      pct.
%d",i);scanf("%d",&p[i].x);
          printf("dati
p[%d].x",i);scanf("%d",&p[i].x);
          printf("dati
p[%d].y",i);scanf("%d",&p[i].y);
          }
          p[0]=p[N];
          p[N+1]=p[1];
          }
}
```

- Desenarea poligonului se face cu ajutorul funcției:

```
void desenare_poligon(int n, punct p[200])
{
     char c[3];
     moveto( p[n-1].x,p[n-1].y);
     for (i=0;i<n;i++){
        lineto(p[i].x,p[i].y);
        outtextxy(p[i].x+10,p[i].y, p[i].nume);
         }
     }
```

5.2 Orientarea triungiurilor şi testul de coliniaritate

Prima problemă de geometrie computațională revine la a extinde, într-o anumită manieră, relația de ordine a numerelor de pe dreapta reală la o mulțime finită de puncte în plan (minim trei puncte). Astfel, introducem o relație geometrică care operează cu puncte într-o manieră analogă cu operațiile relaționale clasice (<,=,>) cu numere. Aparent nu există nici un mod natural intrinsec de a a compara două puncte în spatiul *d*-dimensional (*d*=2,3), dar există o legătură naturală între puncteledin spațiul *d*-dimensional care extinde noțiunea de relații binare din spațiul unidimensional, legătură numită **orientare**.

5.2.1 Orientarea punctelor

Fiind dat tripletul de puncte (***p, q, r***) în plan, spunem că au *orientare pozitivă* dacă definesc un triunghi orientat antiorar, adică trigonometric, *orientare negativă* dacă definesc un triunghi orientat în sens orar, şi *orientarea de zero* dacă se află pe o dreaptă comună (care include de asemenea cazul în care două sau mai multe puncte sunt identice). Evident orientarea depinde de ordinea în care se dau punctele.

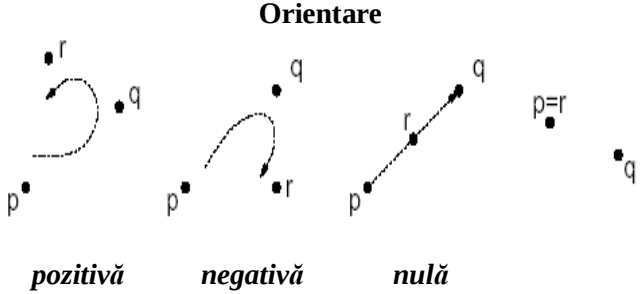


Fig. 5.1 Orientarea a trei puncte

Definiție 5.1

Orientarea unui număr de *d*+1 puncte în spațiul d-dimensional este definită formal de semnul determinantului coordonatelor omogene ale punctelor date. În cazul unidimensional, *Orientare(p,q)* este tocmai *q-p*, deci orientarea este pozitivă dacă p < q, zero dacă *p* = *q*, şi negativă dacă *p* > *q*.

În plan, avem

$$Orientare(p,q,r) = sign \det \begin{pmatrix} 1 & p.x & p.y \\ 1 & q.x & q.y \\ 1 & r.x & r.y \end{pmatrix} \qquad (5.2)$$

În C avem funcția de orientare a unui triunghi:

```
int orientare(punct P,punct Q,punct R){
      float d=(float)(Q.x-P.x)*(R.y-P.y)
                  -(float)(Q.y-P.y)*(R.x-P.x);
      if (d>0) return 1;
      if (d<0) return –1;
if (d==0) return 0;
```

Funcția returnează valoarea 1 dacă ΔPQR este orientat pozitiv, -1 dacă este orientat negativ şi 0 dacă cele trei puncte sunt coliniare.

De asemenea se observă că semnul de orientare al unui triplet de puncte este neschimbat dacă punctele sunt translatate, rotite, sau scalate (cu un factor de scală pozitiv). O transformare de simetrie, de exemplu $(x,y)\rightarrow(-x,y)$, schimbă semnul orientării! Putem spune că, *aplicând unui punct orice transformare afină, se schimbă semnul orientarii conform cu semnul matricii folosită în transformare.*

În general, când se dau coordonatele a 4 puncte în spatiul tridimensional, putem de asemenea să le definim orientarea ca fiind ori pozitivă (formând o răsucire spre dreapta), ori negativă (o răsucire

spre stânga), sau zero (coplanare). Aceasta se poate generaliza la orice puncte de coordonate (d+1)-tuple în spațiul d-dimensional.

Pe de altă parte, o orientare nulă a unui triunghi se traduce prin coliniaritatea vârfurilor sale. Coliniaritatea a trei puncte din plan se poate determina deci prin apelarea funcției :

```
int coliniare(punct A,punct B,punct C)
{
      if(orientare(A,B,C)!=0) return 0;
            else return 1;
}
```

Daca trei puncte sunt coliniare, este util uneori să cunoaştem poziția lor pe dreapta suport. Funcția următoare determină dacă punctul B se găsește între punctele A şi C:

```
   int intre(punct A,punct B,punct C)
      {
      if(!coliniare(A,B,C))  return 0;
      if(A.x<C.x)       return

      (((A.x<=B.x)&&(B.x<=C.x))||((A.x>=B.x)&&(B.x
>=C.x)));
         else     return

      (((A.y<=B.y)&&(B.y<=C.y))||((A.y>=B.y)&&(B.y
>=C.y)));
      }
```

5.2.2 Suprafețe şi unghiuri

Orientarea determinantului, împreună cu norma Euclidiană pot fi folosite pentru a calcula unghiurile în plan. Determinantul

$$Or(A,B,C) = \det\begin{pmatrix} 1 & A.x & A.y \\ 1 & B.x & B.y \\ 1 & C.x & C.y \end{pmatrix} \tag{5.3}$$

este egal cu de două ori suprafața marcată a triunghiului Δ**ABC** (pozitiv dacă avem o orientare trigonometrică şi negativ în celălalt

caz). Astfel suprafața triunghiului poate fi determinată împărțind această mărime la 2. (În general, în *d* dimensiuni, hyper-volumul simplexului definit de *d*+1 puncte poate să fie determinat luând determinantul coordonatelor omogene ale punctelor şi împărțindu-l la ***d!***). Implementarea funcției de arie în C este:

```
   float aria(punct A,punct B,punct C){
      float           d=(float)(B.x-A.x)*(C.y-A.y)-
(float)(B.y-A.y)*(C.x-A.x);
      return fabs(d/2);
      }
```

iar aria unui poligon se poate obține prin împărțirea lui în triunghiuri şi însumarea ariilor acestora.

Produsul scalar, după cum s-a văzut în capitolul 3, defineşte cosinusul unui unghi a doi vectori. Oricum, acest lucru nu este folositor pentru a deosebi unghiurile pozitiv orientate de cele negativ orientate. Sinusul unghiului $\theta=\angle$**ABC** (unghiul definit de vectorul **A-B** şi vectorul **C-B**) poate fi calculat astfel

$$\sin\theta = |AB|\,|CB|\,Or(B, A, C) \qquad (5.4)$$

(Ordinea parametrilorde mai sus este esențială, funcția *sinus* fiind negativă pentru o orientare a unghiului θ în sensul acelor de ceasornic şi pozitivă pemtru o orientare triginometrică a unghiului.)

5.3 Intersecții proprii şi improprii de segmente

Ca şi în cazul exemplului precedent, vom considera problema elementară de geometrie a determinării intersecției a două segmente de dreaptă date. Figura 5.2 ilustrează două exemple de intersecții care pot apărea:

Cel mai curent caz de intersecție este exemplificat prin poziția segmentelor [CD] și [BG], când punctul comun aparține interiorului ambelor segmente de dreaptă, dar, pe de altă parte, trebuie să ținem cont și de eventualitatea ca unul din capetele unui segnent să aparțină celuilalt segment, ca și în cazul segmentelor [KP] și [IL].

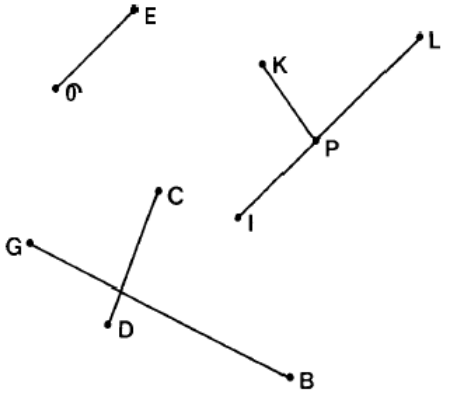


Figura 5.2 Intersecții de segmente

Definiție 5.2

Fie A, B, C, D patru puncte din plan. Dacă $[AB] \cap [CD] \neq \Phi$, spunem că intersecția este *proprie* dacă aceasta nu conține nici unul dintre punctele A, B, C, D; spunem ca intersecția este *improprie* dacă ea conține cel puțin unul dintre punctele A, B, C, D.

Existența unei intersecții proprii a două segmente se poate obține cu funcția:

```
int intersecție_proprie(punct A, punct B,
punct C, punct D){
      return ((orientare(A,B,C) *
orientare(A,B,D)<0) &&  (orientare(C,D,A) *
orientare(C,D,B) < 0));
}
```

Existența unei intersecții improprii a două segmente se poate obține cu funcția:

```
int intersecție_improprie(punct A,
punct B, punct C, punct D){
      return ((intre(C,A,D)) ||
(intre(C,B,D)) ||
(intre(A,C,B)) || (intre(A,D,C)));
}
```

Intersecția a doua segmente este nevidă dacă este proprie sau improprie:

```
int intersecție_segmente(punct A, punct B,
punct C, punct D){
      return ((intersecție_proprie(A,B,C,D))
      ||intersecție_improprie(A,B,C,D)));
}
```

Dacă cele două segmente nu au puncte comune, considerăm intersecția dreptelor definite de cele 2 segmente (în cazul în care aceasta există, adică segmentele nu sunt paralele, ca şi [OE] şi [IL] din figura 4.2). Punctul de intersecție a celor două drepte poate să aparțină unuia dintre segmente (doar unuia!), ca şi în cazul [IL] şi [BG], poate fi exterior ambelor segmente, ca şi în cazul [OE] şi [BG], sau putem avea cazul de excepție când ambele segmente sunt pe aceeaşi dreaptă, caq care se tratează separat. În general **problema determinării poziției relative a celor două segmente** se reduce la determinarea punctului de intersecție a dreptelor-suport ale

segmentelor şi verificarea poziției acestuia pe fiecare dreaptă în parte, relative la capetele segmentelor.

Situația devine cu atât mai complicată cu cât avem un număr mai mare de linii. Mai târziu vom vedea că există un algoritm sofisticat rapid care determină câte dintre perechile dintr-un număr de N de linii se va intersecta.

5.4 Problema drumului simplu închis

Problema drumului simplu închis revine la a găsi linia care trece printr-un nrumăr de N puncte date, nu se intersectează cu ea însăşi, trece prin toate punctele şi se întoarce la punctul de la care a plecat. O astfel de linie frântă se numeşte **drum simplu închis**. Ne putem imagina foarte multe aplicații pentru un astfel de model: punctele pot reprezenta casele, iar linia drumul pe care îl urmeaza poştaşul pentru a ajunge la fiecare casă fără să-şi traverseze drumul deja efectuat. Problema descrisă mai sus este o problemă elementară pentru că se referă determinarea unei modalități oarecare de a uni punctele. A găsi cea mai bună soluție, adică *cel mai scurt drum*, este cu mult mai difícil; pe de altă parte, în capitolul următor ne vom ocupa de o problemă asemănătoare: determinarea celui mai scurt drum care înconjoară un număr de N puncte date.

Cu titlu de exemplu, fie ansamblul de puncte definit la subcapitolul 4.1:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
x:	3	11	6	4	5	8	1	7	9	14	10	16	15	13	2	12
y:	9	1	8	3	15	11	6	4	7	5	13	14	2	16	12	10

O cale de a soluționa problema găsirii unui drum simplu închis este următoarea: alegem un punct care să fie originea reperului

fix; din fiecare punct dat trasăm o linie dreaptă până în punctul origine, apoi unim printr-un set de linii punctele date, *ordonate* în sensul rotirii privirii din punctul de reper. Rezultatul va fi o linie simplă, închisă, ce uneşte toate punctele. În acest exemplu, B este originea, iar punctele date se ordonează în sens trigonometric astfel: M,J,L,N,P,K,F,I,E,C,O,A,H,G,D. Unindu-le acum printr-o linie frântă închisă ce pleacă din B, obținem un poligon.

Dacă dx şi dy sunt distanțele pe axele *x* şi *y* de la un punct dat oarecare la origine, atunci unghiul pe care-l face segmentul ce uneşte punctul considerat cu originea față de orizontală (axa Ox) de care avem nevoie în acest algoritm este tan^{-1}(dy/dx). Din moment ce acest unghí este doar formal folosit în algoritm, este mai indicat să folosim o funcție pentru o ordonare asemănătoare, cu care vom obține acelaşi rezultat. O astfel de funcție poate fi considerată dy/(dy+dx). Chiar şi aşa, funcția trebuie testată, dar testarea va fi mult mai simplă.

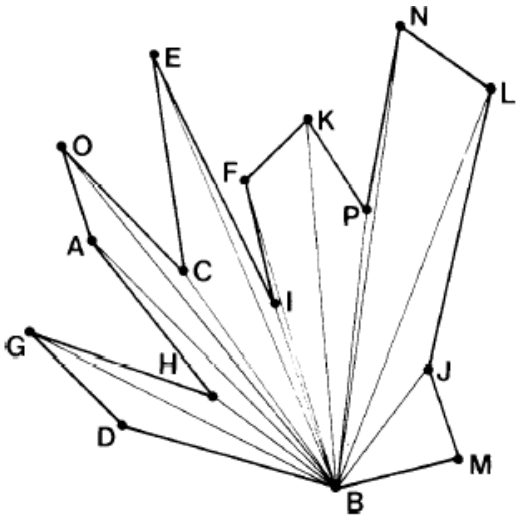


Fig. 6.3 Un drum simplu închis ce leagă punctele din exemplul de mai sus

Următorul program întorce un număr între 0 şi 360 care nu este unghiul dintre [AB] şi orizontală, dar are aceleşi proprietăți de ordonare.

```
float theta(punct A, punct B)
  {int x,y,ax,ay;
   float t;
      x=B.x-A.x; ax=fabs(x);
      y=B.y-A.y; ay=fabs(y);
      if ( (x==0)&&(y==0)) t=0;
            else t=y/(ax+ay);
      if (x<0) t=2-t;
      if (y<0) t=4-t;
      return t*90.0;
}
```

În sfârşit, ordonarea punctelor în raport cu „originea” (punctul de ordonată *y* cea mai mică) se efectuează cu o simplă funcție de sortare.

5.5 Apartenența la interiorul unui poligon

Problema aparteneței unui punct dat la interiorul unui poligon este o problemă uşoară vizual şi ceva mai complicată din punct de vedere algoritmic. Dacă în cazul unui triunghi lucrurile sunt simple (pentru triunghiuri orientate trigonometric este suficient ca punctul să se situeze la stânga oricărei laturi, iar pentru cele orientate negativ, puntul să se regăsească la dreapta fiecărei laturi), în cazul poligoanelor neconvexe situația poate să se complice mai mult.

Pentru trei puncte necoliniare A,B,C funcția următoare verifică dacă un punct M se găseşte în interiorul triunghiului ABC sau în exteriorul acestuia.

```
int punct_in_triunghi(punct M,punct A,punct
B,punct C){
```

```
if (orientare(A,B,C)>0) return
     ((orientare(M,A,B)>=0) &&
(orientare(M,B,C)>=0)
      && (orientare(M,C,A)>=0));
     if(orientare(A,B,C)<0) return

     ((orientare(M,A,B)<=0)&&(orientare(M,B,C)<=0
)&&

     (orientare(M,C,A)<=0));
}
```

O idee în cazul poligoanelor generale este de a trasa o semi-dreaptă arbitrară prin punctul testat şi de a verifica dacă ea taie laturile poligonului într-un număr par sau impar de ori. dacă numarul de linii este *impar*, punctul trebuie să fie *înăuntrul poligonului*, iar dacă este par, punctual este *exterior*. Acest lucru se poate vedea cu uşurinţă urmărind ceea ce se întâmplă în timp ce ne îndreptăm spre interior pornind de la un punct din exterior: după ce intersectăm prima latură suntem în interiorul poligonului,dupa cea de a doua suntem din în exterior, etc. Dacă se repetă acţiunea de un număr par de ori, punctul la care ajungem (punctul iniţial) trebuie să fie în exterior.

Situatia nu este atat de simplă, deoarece unele intersecţii pot apărea chiar într-un vărf al poligonului. Desenul de mai jos arată câteva din situaţiile care trebuie tratate.

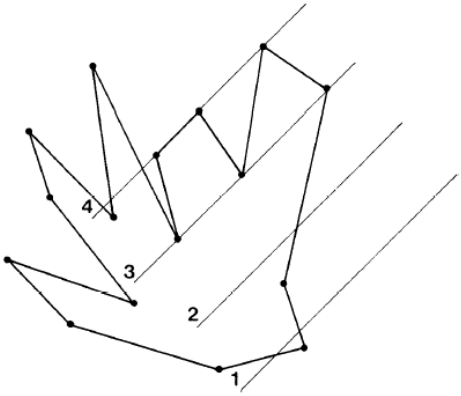


Fig. 5.4 Patru cazuri diferite de intersecţii ale unei drepte cu un poligon

Dreptele 1 şi 2 taie laturile poligonului în interiorul lor, dreapta 3 trece prin trei dintre vârfurile poligonului, iar dreapta 4 cuprinde una dintre laturile poligonului. Punctele-vârfuri pe unde trece drapta 3 vor fi considerate intersecţii *duble* cu poligonul dacă cele două laturi adiacente vârfului *se află de aceeaşi parte a dreptei* 3; şi vor fi considerate simple în caz contrar. În cazul dreptei 4, segmentul de dreaptă comun poate fi considerat ca *un singur vârf fictiv al poligonului*, reducând astfel problema la cazul precedent.

Nevoia de a trata situaţiile în care vârfurile poligonului se intersectează cu dreptele de test ne forţează să facem mai mult decât să numărăm laturile poligonului care se intersecteaza cu dreapta de test. O metoda de a implementa acest lucru este să luăm în calcul intersecţiile cu o dreaptă de test orizontală, definită ca segment foarte larg pe tot domeniul *int*, ca mai jos:

```
int inside(punct M, punct p[N+1]){
      {int i,j,nr;
      dreapta test,lp;
      nr=0; j=0;
      p[0]=p[N]; p[N+1]=p[1];
          \\poligonul este declarat ciclic
      test.p1.x=-32760; test.p2.x=32760;
```

```
test.p1.y=M.y; test.p2.y=M.y;
    \\ se defineste o dreapta orizontala
    \\ ce trece prin M
for (i=1;i<N+1;i++)
   { lp.p1=p[i]; lp.p2=p[i+1];
      if
intersecţie_segmente(test.p1,test.p2,lp.p1,l
p.p2)
                  nr=nr+1; \\ numara
intersectiile
      if (p[i].y==test.y) &&( (p[i+1].y-
      p[i].y)*
            (p[i-1].y- p[i].y)<0 ) nr=nr-1;
\\ cazul 3 din exemplul de mai sus, cand
\\ laturile adiacente NU sunt de aceeasi
\\ parte a dreptei de test
      }
return nr%2;
}
```

Cititorul pote să verifice dacă acest algoritm functionează corespunzător în cazurile 3 şi 4 din figura 4.4.

Din cele câteva exemple date ar trebui să fie clar că este uşor să subestimăm dificultatea rezolvării unei probleme geometrice specifice cu ajutorul calculatorului. Exista multe alte calcule geometrice elementare pe care nu le-am discutat. De exemplu, un exerciţiu interesant ar fi determinarea suprafeţei unui poliedru neregulat cu ajutorul unui program. Totuşi problemele elementare precedente ne-au oferit câteva instrumente de bază pe care le vom găsi utile în capitolele urmatoare, în soluţionarea unor probleme mai dificile.

Unii dintre algorimii studiaţi conduc la construirea unor structuri geometrice speciale pentru un set dat de puncte. ”Linia poligonală închisă simplă” este un exemplu elementar pentru asta.

Puțini algoritmi geometrici au fost analizați până la nivelul la care se pot face afirmații exacte legate de caracteristicile relative de performanta ale lor. Complexitatea unui algoritm geometric poate depinde de multe lucruri. Distribuția însăşi a punctelor, ordinea în care ele apar, dacă funcțiile trigonometrice sunt necesare sau pot fi înlocuite cu alte funcții mai simple, toate acestea pot să afecteze semnificativ timpul de execuție al algoritmilor geometrici. Ca de obicei în asemenea situații, avem dovezi empirice care sugerează alegerea algorimilor potriviți pentru aplicații specifice. De asemenea, mulți dintre algoritmi sunt meniți să genereze rezultate bune chiar în situațiile cele mai defavorabile.

5.6. Teme de laborator

1. Implementați algoritmul descris la 5.3 pentru a verifica existența unor intersecții de segmente.

Soluție

```
#include <graphics.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>

typedef struct { float x,y; }punct;

int orientare(punct A,punct B,punct C){
            float     d=(float)(B.x-A.x)*(C.y-A.y)-
(float)(B.y-A.y)*(C.x-A.x);
      if (d>0) return 1;
      if (d<0) return 1;
      if (d==0) return 0;
}

int coliniare(punct A,punct B,punct C){
```

```
      if(orientare(A,B,C)!=0) return 0;
            else return 1;
}
int intre(punct A,punct B,punct C){
        if(!coliniare(A,B,C))  return 0;
        if(A.x<C.x)                              return
(((A.x<=B.x)&&(B.x<=C.x))||((A.x>=B.x)&&(B.x>=C.x)
));
              else
return(((A.y<=B.y)&&(B.y<=C.y))||((A.y>=B.y)&&(B.y
>=C.y)));
      }

int intersectie_proprie(punct A, punct B, punct C,
punct D){
return  ((orientare(A,B,C)  *  orientare(A,B,D)<  0)
&& (orientare(C,D,A) * orientare(C,D,B) < 0));
}

int  intersectie_improprie(punct  A,  punct  B,  punct
C, punct D){
      return  ((intre(C,A,D))  ||  (intre(C,B,D))  ||
(intre(A,C,B)) || (intre(A,D,C)));
}

int  intersectie_segmente(punct  A,  punct  B,  punct
C, punct D){
      return   ((intersectie_proprie(A,B,C,D))   ||
(intersectie_improprie(A,B,C,D)));
}

int main()  {

      punct A,B,C,D;
      printf("coordonatele punctului A sunt:");
      scanf("%f%f", &A.x,&A.y);
      printf("coordonatele punctului B sunt:");
      scanf("%f%f", &B.x,&B.y);
      printf("coordonatele punctului C sunt:");
      scanf("%f%f", &C.x,&C.y);
      printf("coordonatele punctului D sunt:");
```

```
      scanf("%f%f", &D.x,&D.y);

      initwindow(800,600, "intersectii ",200,200);

      setcolor(14);
      outtextxy(A.x+5,A.y+5,"A");
            outtextxy(B.x+5,B.y+5,"B");
                  outtextxy(C.x+5,C.y+5,"C");

      outtextxy(D.x+5,D.y+5,"D");


    line(A.x,A.y,B.x,B.y);
      setcolor(12);
      line(C.x,C.y,D.x,D.y);
      if(intersectie_segmente(A,B,C,D))
printf("intersectie nevida");
            else printf("intersectie vida");
      getch(); getch();
closegraph();

return 0;
}
```

2. Implementați algoritmul descris la 5.5, folosind un poligon cu vârfuri generate aleator şi algoritmul de construcție a unei linii poligonale simple dat mai jos. Coordonatele punctului de test se vor introduce de la tastatură.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
#include<stdlib.h>
using namespace std;

typedef struct{
      float x,y;
```

```
      }punct;


int n; //numarul de puncte
      punct p[100];

typedef struct{
      float x,y;
}vector;


float produs_scalar(vector u,vector v){
            return (u.x*v.x+u.y*v.y);
            }

float cos_unghi(vector u, vector v){
     return
(produs_scalar(u,v)/(sqrt((produs_scalar(u,u))*(pr
odus_scalar(v,v)))));
     }

int jos_stanga(){
      int imin=1,i;
      for(i=1;i<=n;i++){

      if((p[i].y<p[imin].y)||((p[i].y==p[imin].y)&
&(p[i].x<p[imin].x)))
            imin=i;
            }
      return imin;
      }



void initializare(){
  printf(" dati  nr. de puncte n=");
  scanf("%d",&n);
//PLEACA DE FIECARE DATA DIN ALT PUNCT
   for(int i=1;i<=n;i++){
         p[i].x=rand()%500 + 50;
             p[i].y=rand()%500+ 143;
      }
}


void desenare(){
            char c[3];
```

```
            int i;
            setcolor(1);

            for(i=1;i<=n;i++){
                  itoa(i,c,10);
                  setcolor(1);
                  circle(p[i].x,p[i].y,2);
                  setcolor(4);
                  outtextxy(p[i].x,p[i].y,c);
                  }
            }


void drum(){
      punct aux;
      float cos_max;
      vector dc,vi, viplus1;
      int ord,i;
      int imin=jos_stanga();
      dc.x=1;
      dc.y=0;


     aux.x=p[imin].x;   aux.y=p[imin].y;
    p[imin].x=p[n].x;  p[imin].y=p[n].y;
    p[n].x=aux.x; p[n].y=aux.y;

     p[0].x=p[n].x;  p[0].y=p[n].y;
   printf("\n Pozitia minimului a fost %d", imin);

   //ordonarea punctelor
   do { ord=1;
   for (i=1;i<n-1;i++)
       {  vi.x= p[i].x-p[0].x;
          vi.y= p[i].y-p[0].y;
          viplus1.x= p[i+1].x-p[0].x;
          viplus1.y= p[i+1].y-p[0].y;
          if  (cos_unghi(vi,dc)<  cos_unghi(viplus1,
dc) )
                              { ord=0;
                                aux.x=p[i].x;
      aux.y=p[i].y;
                                p[i].x=p[i+1].x;
p[i].y=p[i+1].y;
                                p[i+1].x=aux.x;
p[i+1].y=aux.y;
                                   };
```

```
                };
           }
     while (ord==0);

   //afisare drum
   setcolor(2);
  for (i=0;i<n;i++)
    line(p[i].x,p[i].y, p[i+1].x,p[i+1].y);
}

void afisare(){
    for(int i=1;i<=n;i++)
      cout<<"Punctul "<<i<<" are coord: "<<p[i].x
<<",\t"<<p[i].y<<"\n";
     }

int  main(){
initializare();
//int gd=DETECT,gm;
//clrscr();
initwindow(800,800, "Drum simplu inchis");

setbkcolor(15);
cleardevice();
desenare();
afisare();
drum();

getch();
closegraph();
      }
```

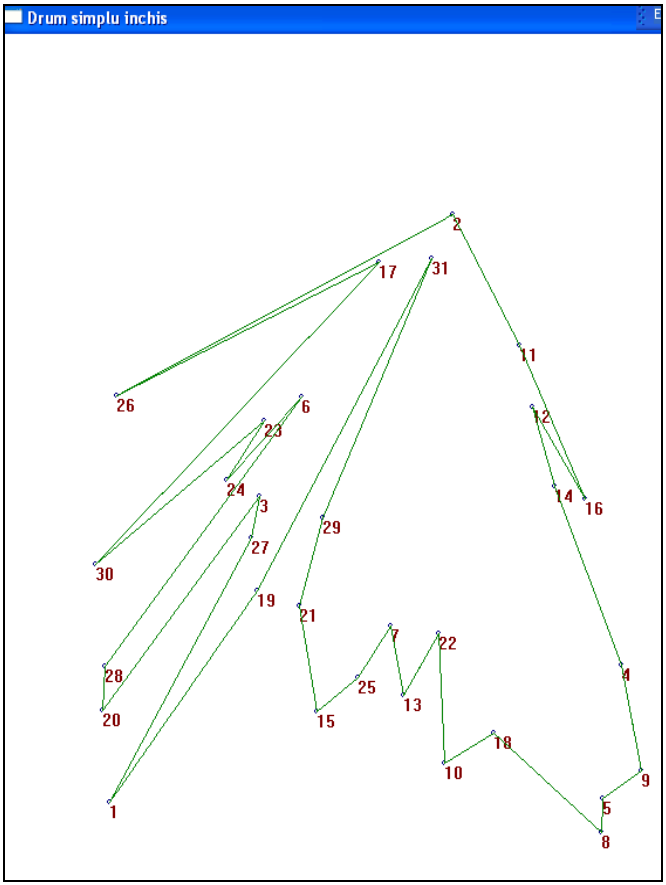


Fig. 5.5 Execuția programului precedent

Capitolul VI. Probleme de intersecții

Descriere Generală

În acest capitol vom studia metoda generală pentru determinarea intersecților de segmente de dreaptă dintr-un ansamblu de N segmente, folosind algorimi de complexitate proportională cu N·log(N), bazat pe metoda lui M. Shamon şi D. Hoey (1976, [7]).

Obiective

- Introducerea noțiunii de "Geometrie Manhattan".
- Descrirerea algoritmilor de scanare orizontală a segmentelor de dreaptă.

6.1. Intersecții geometrice

Una dintre problemele de bază în geometria computațională este aceea de a determina intersecțiile de diverse obiecte geometrice. Calcularea intersecțiilor în spatiul bi şi tridimensional este de bază în diferite zone de aplicație.

- Într-un sistem pentru desenarea şi procesarea circuitelor integrate sau tipărirea tabelelor-circuit, este important de ştiut dacă se ating doua legături electrice pentru a nu face un scurt circuit.
- În robotică şi planificarea în mişcare este important de ştiut când două obiecte se intersectează în *detectarea coliziunii* şi *evitarea coliziunii*.
- În sistemele informaționale geografice este adesea folositor să suprapunem două subdiviziuni (e.x. o rețea rutieră şi granițele județelor pentru a determina răspunderile pentru întreținerea drumurilor). Deoarece aceste rețele sunt formate din grupuri de segmente de dreaptă, acest lucru generează o problemă de determinare a intersecției segmentelor de dreaptă.
- În grafica pe calculator, problema de a determina care set de obiecte este obscurat dintr-un unghi de vedere particular poate fi formulată ca o problemă de intersecție geometrică în proiectarea obiectelor în planul vizual.

Solutia evidenta a problemei intersecției este să verificăm fiecare parte a obiectelor şi să vedem dacă ele se intersecteaza. Daca avem N obiecte geometrice, timpul necesar acestui algoritm naiv este proportional cu N^2. Pentru multe aplicatii practice aceasta nu este o problemă, deoarece alți factori limitează numărul de obiecte care trebuie procesat. Oricum, aplicațiile geometrice actuale a devenit mult mai ambițioase, şi nu este ieşit din comun să avem de studiat sute de mii sau chiar milioane de obiecte. O complexitate algoritmica proporțională cu N^2 este în mod evident inadecvată pentru asemenea aplicații.

În acest capitol vom studia metoda generală pentru determinarea intersecților de segmente de dreaptă dintr-un ansamblu de N segmente, folosind algorimi de complexitate proportională cu N·log(N), bazat pe metoda lui M. Shamon şi D. Hoey (1976, [7]).

6.2. Intersecții de segmente de dreaptă

Intersecția segmentelor de dreaptă: Problema pe care o vom lua în considerație este următoarea: *se dau **n** segmente de dreaptă în plan, aflați toate punctele în care se intersectează două segmente de dreaptă.* Presupunem că fiecare segment de dreaptă este reprezentat de coordonatele celor două puncte terminale.

A se observa că cele *n* segmente de dreaptă se pot intersecta în cel puțin 0 şi cel mult $\binom{n}{2}$ puncte diferite. Am putea să folosim un algoritm cu timp $O(n^2)$, pretinzând că cel mai rău-caz este asimptotic optim, dar el nu este foarte folositor în practică, deoarece în multe exemple de probleme de intersecție, intersecțiile propriu-zise sunt rare. În consecință este rezonabil să căutam un *algoritm cu rezultat de precizie*, unul al cărui timp de rulare ar trebui să fie mai eficient

decât determinarea pur geometrică a tuturor intersecțiilor de segmente.

Fie *I* numărul de intersecții. Vom presupune că segmentele de dreaptă se află în poziție generală, şi astfel nu ne vom preocupa de problema dacă trei sau mai multe drepte se intersectează într-un singur punct. Totuşi, generalizarea algoritmului pentru a manipula asemenea degenerări în mod eficient este un exercițiu interesant.

6.2.1 Intersecția brută a segmentelor de dreaptă

Ca multe alte funcții geometrice, calcularea punctului în care se intersectează două segmente de dreaptă poate fi redusă la rezolvarea unui sistem de 2 ecuații liniare. Avem *AB* şi *CD*, două segmente de dreaptă, date de punctele lor terminale. Mai întâi se observă că este posibil să determinăm dacă aceste segmente de dreaptă se intersectează, pur şi simplu aplicând o combinație corespunzătoare de teste de orientare, la fel ca în secțiunea 4.3.

O metodă de a determina punctul în care segmentele se intersectează este aceea de a folosi un truc vechi din sistemul de calcul al graficelor. Vom reprezenta segmentele folosind o reprezentare parametrică: orice punct *P* de pe segmentul de dreaptă *AB* poate fi descris ca o combinație liniară afină de puncte *A* şi *B*, introducând un parametru real *t*:

$x_P(t) = (1-t)x_A + t\,x_B$ pentru $0 \le t \le 1$.
$y_P(t) = (1-t)y_A + t\,y_B$ pentru $0 \le t \le 1$.

În mod similar pentru un punct *Q* de pe CD putem introduce un parametru *s*:

$x_Q(s) = (1-s)x_C + s x_D$ pentru $0 \le s \le 1$.
$y_Q(s) = (1-s)y_C a + s\,y_D$ pentru $0 \le s \le 1$.

.O intersecție are loc dacă şi numai dacă punctele P şi Q sunt identice. Astfel, obținem următoarele două ecuații:

$(1-t)xA + t\,xB = (1-s)xC + s\,xD$

$(1-t)yA + t\,yB = (1-s)yCa + s\,yD$.

Coordonatele punctelor ne sunt cunoscute, astfel doar printr-un exercițiu simplu de algebră liniară putem afla parametrii s şi t. Calculul lui s şi t implică o împărțire. Dacă împărțitorul este 0, atunci segmentele de dreaptă sunt paralele sau coliniare. Dacă împarțitorul este diferit de 0, atunci vom obține valori pentru *s* şi *t*, numere raționale (rație a doi întregi. Odată ce valorile lui *s* şi *t* au fost calculate tot ce rămâne de făcut este să se verifice dacă valorile se află în intervalul [0,1].

6.2.2 Linii orizontale si verticale

Pentru început, fie cazul în care segmentele sunt sau orizontale, sau verticale: doua puncte definesc fiecare segment, ele avand coordonatele x egale, sau coordonatele y egale, ca în Fig.6.1. (O astfel de mulțime de segmente poartă numele de geometrie Manhattan, din cauza preferinței arhitecților New York-ului pentru linii orizontale şi verticale perpendiculare.)

Constrângând segmentele să fie orizontale sau verticale impune în mod sigur o severă restricție, dar acest lucru este departe de a simplifica problema prea mult. Avem cazuri în care această restricție este impusă pentru o aplicație particulară. De exemplu, multe planificări de circuite integrate sunt tipic desenate respecând o astfel de constrângare.

Ideea generală a unui algoritm pentru determinarea intersecțiilor într-un asemenea ansamblu de segmente este de a imagina o scanare orizontală. Proiectand toate segmentele pe o dreaptă orizontală, cele verticale devin puncte, iar cele orizontale devin intervale: când segmentele sunt scanate de jos în sus, punctele (reprezentând segmente verticale) apar şi dispar, iar segmentele orizontale se întalnesc sporadic. O intersecție este presupus determinată atunci când dreapta de scanare orizontală se întalneşte cu un punct reprezentând o linie verticala. Intersecția propriu-zisă inseamnă că segmentul vertical intersectează segmentele orizontale scanate în acel moment.

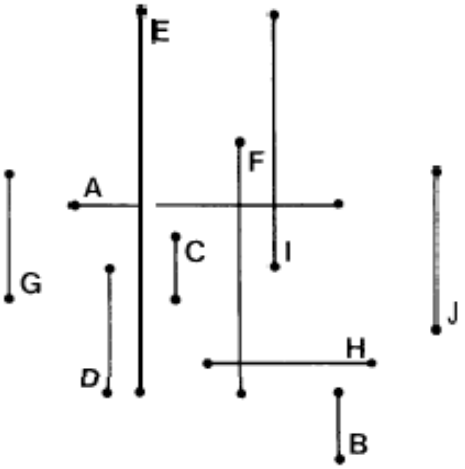


Figura 6.1 Geometria Manhattan

Ca reprezentare internă, mai întâi se generează o lista ordonată de segmente **după coordonata verticală *y*** a extremităților segmentelor, unde fiecare segment vertical este reprezentat de **două** ori (două extremități de coordonată y distincte) iar cele orizontale doar o dată. Aceasta se realizează printr-o ordonare rapidă a tuturor coordonatelor y distincte, fiecare nouă înregistrare în lista ordonată păstrând pe lângă informație de identificare a segmentului şi o informație de tipul 0-„segment vertical”, 1-„segment orizontal”.

Pentru exemplul din Figura 6.1, lista ordonată ar trebui să arate astfel

BBDEFHJCGDICAGJFEI

O simplă parcurgere a acestei liste de la stânga la dreapta simulează o scanare verticală a ansamblului de segmente. De aici, parcurgând lista prealabilă de mai sus, la fiecare moment se reţin într-un buffer (modelat eventual printr-o coadâ FIFO) simbolurile acelor segmente verticale pentru care coordonata inferioară a fost

deja întâlnită, iar cea superioară încă nu. Pentru fiecare simbol de segment orizontal întâlnit, caracterizat de „x-intervalul" coordonatelor orizontale ale extremităților sale, aparteneța coordonatei x a oricărui segment vertical - *aflat în buffer* la acel moment! – la x-intervalul corespunzător determină o intersecție de segmente.

Pe exemplul de mai sus, bufferul va arăta succesiv astfel

B
D
DE
DEF

iar în acest moment se întâlneşte simbolul segmentului orizontal H, care deci poate să intersecteze cel mult segmentele D, E şi F. Pentru cazul secmentului A, bufferul evoluează în continuare astfel:

DEFJ
DEFJC
DEFJCG
EFJCG
EFJCGI
EFJGI

iar în acest moment se întâlneşte segmentul A, care poate intersecta doar segmentele din buffer E,F,J,G,I.

Evident, problema a două segmente orizontale care au puncte comune se poate rezolva destul de uşor în acest context, întrucât ele vor avea aceeaşi coordonată verticală şi vor fi vecine în lista pre-odonată de la primul pas. Problema a două segmente verticale care au puncte comune este mai puțin evidentă şi trebuie tratată separat**.**

6.2.3 Intersecții de segmente arbitrare

Problema determinării intersecțiilor de segmente se modifică în momentul în care orientarea acestora este arbitrară (vezi figura de

mai jos)

Primul pas în acest caz va fi tot generarea unei liste pre-ordonate de segmente, în funcție de coordonatele z ale lor, dar fiecare segment este astfel reprezentat prin 2 extremități, fiind prezent de **două ori în listă**. Pentru exemplul de mai sus, lista inițială ar fi de forma

BHCCGDBAEEAFHDG

Ideea de scanare verticală este similară cu cea din cazul geometriei Manhattan, prin parcurgerea listei de la stânga la dreapta şi reținerea într-un buffer a tuturor segmentelor ale căăror extremități inferioare au fost deja „întâlnite" şi pentru care extremitățile superioare du au fost încă „atinse".

Fiecare segment „nou" întâlnit este mai întâi testat de eventualele intersecții cu segmentele din buffer, prin procedura descrisă la secțiunea întâia a acestui sub-capitol, apoi introdus la rândul lui în buffer. Un segment prezent în buffer care este întâlnit a doua oară va fi şters. Astfel se testează posibilele intersecții ale segmentului C doar cu segmentele B şi H, iar ale segmentului A doar cu H, G şi D.

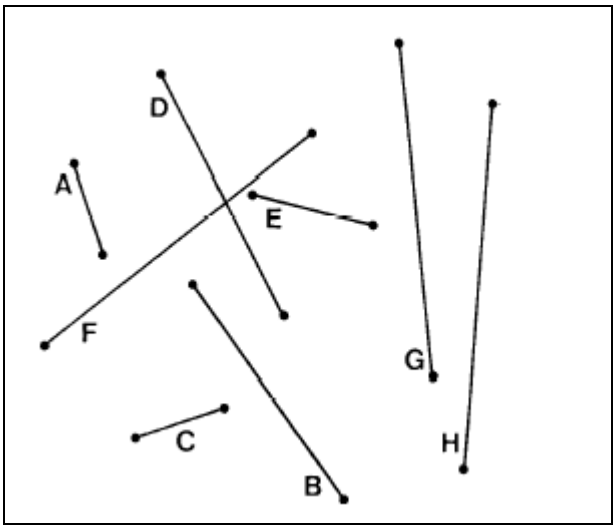


Figura 6.2 Cazul general

Metoda poate fi adaptată pentru a reduce intersecțiile triple sau multiple de segmente, intersecții care altfel sunt numărate de mai multe ori.

6.3. Teme de laborator

1. Implementați algoritmul descris la 8.2.2 pentru a număra câte intersecții de segmente sunt pe ecran, modificând următoarea implementare “naivă”:

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
#include<stdlib.h>
using namespace std;

typedef struct {
 float x,y;
```

```
 }punct;

typedef struct{
      int x1,y1,x2,y2;
      }segment;

int n; //numarul de segmente
      segment p[100];

int orientare(punct A,punct B,punct C){
            float d=(float)(B.x-A.x)*(C.y-A.y)-
(float)(B.y-A.y)*(C.x-A.x);
      if (d>0) return 1;
      if (d<0) return -1;
      if (d==0) return 0;
}

int coliniare(punct A,punct B,punct C){
      if(orientare(A,B,C)!=0) return 0;
            else return 1;
}
int intre(punct A,punct B,punct C){
        if(!coliniare(A,B,C))  return 0;
        if(A.x<C.x) return
(((A.x<=B.x)&&(B.x<=C.x))||((A.x>=B.x)&&(B.x>=C.x)
));
             else
return(((A.y<=B.y)&&(B.y<=C.y))||((A.y>=B.y)&&(B.y
>=C.y)));
      }
int punct_in_triunghi(punct M,punct A,punct
B,punct C){
       if (orientare(A,B,C)>0) return
((orientare(M,A,B)>=0) && (orientare(M,B,C)>=0)
&& (orientare(M,C,A)>=0));
 if(orientare(A,B,C)<0) return
((orientare(M,A,B)<=0)&&(orientare(M,B,C)<=0)&&

      (orientare(M,C,A)<=0));
}
int intersectie_proprie(punct A, punct B, punct C,
punct D){
return ((orientare(A,B,C) * orientare(A,B,D)< 0)
&& (orientare(C,D,A) * orientare(C,D,B) < 0));
}
int intersectie_improprie(punct A, punct B, punct
C, punct D){
```

```
      return ((intre(C,A,D)) || (intre(C,B,D)) ||
(intre(A,C,B)) || (intre(A,D,C)));
}

int intersectie_segmente(punct A, punct B, punct
C, punct D){
      return ((intersectie_proprie(A,B,C,D)) ||
(intersectie_improprie(A,B,C,D)));
}


void initializare(){
  printf(" dati  nr. de segmente n=");
  scanf("%d",&n);
//PLEACA DE FIECARE DATA DIN ALT PUNCT
   for(int i=1;i<=n;i++){
         p[i].x1=rand()%500 + 50;
             p[i].y1=rand()%500+ 143;
             p[i].x2=rand()%500 + 50;
             p[i].y2=rand()%500+ 143;
      }
}


void desenare(){
            char c[3];
            int i,color;
            setcolor(LIGHTBLUE);

            for(i=1;i<=n;i++){
                  itoa(i,c,10);
                  color=rand()%15;
                  setcolor(color);
                  circle(p[i].x1,p[i].y1,2);
                  circle(p[i].x2,p[i].y2,2);
                  outtextxy(p[i].x2,p[i].y2,c);

      line(p[i].x1,p[i].y1,p[i].x2,p[i].y2);
                  }
            }

int intersectii()
{  punct A,B,C,D;
  int contor=0,i,j;

  for(i=1;i<n;i++)
    for(j=i+1;j<=n;j++)
```

```
  {A.x=p[i].x1 ;   A.y=p[i].y1 ;
   B.x= p[i].x2;  B.y= p[i].y2;
   C.x=p[j].x1 ;   C.y=p[j].y1 ;
   D.x= p[j].x2;  D.y= p[j].y2;


  if(intersectie_segmente( A,  B, C, D)) contor++;
  };

 printf("Numarul de intersectii este %d",contor);
 }



int  main(){
initializare();
//int gd=DETECT,gm;
//clrscr();
initwindow(800,800, "intersectii segmente");

setbkcolor(15);
cleardevice();
desenare();
intersectii();


getch();
closegraph();
      }
```

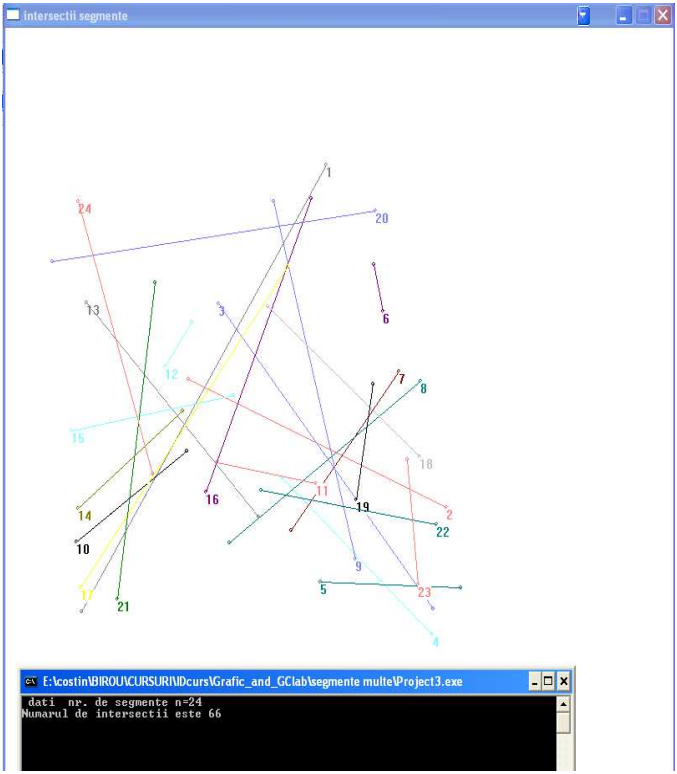


Figura 8.3 Ecranul de execuție al programului precedent

Capitolul VII. Înfăşurătoarea convexă

Descriere Generala

Problematica determinării acoperirilor convexe şi a frontierei acestora (înfăşurătoarea convexă), provenind din geodezie şi topografie, cunoaşte un număr mare de abordări, care au condus la soluţii radical diferite, in funcţie de interpretarea geometrică aleasă. Patru dintre aceşti algoritmi sunt prezentaţi aici.

Obiective

- Introducerea noţiunilor de acoperire convexă şi înfăşurătoare convexă.
- Descrierea algoritmilor de determinare a înfăşurătorii convexe: algoritmul naiv, WrappingHull, Graham Scan şi QuickHull.

Capitolul de faţă se bazează pe lucrarea originală a lui R.L. Graham: *„An efficient algorithm for determining the convex hull of a finite planar set* [4], pe Capitolul 25 din R. Sedgewick – *Algorithms* [7] şi Lecturile 3 şi 4 din David M. Mount - *CMSC 754 Computational Geometry* [5].

7.1 Problema determinării acoperirii convexe şi algoritmul naiv

Există o mulţime de motive pentru care înfăşurătoarea convexă a unei mulţimi de puncte este o structură geometrică importantă. Înfăşurătoarea convexă este una dintre cele mai simple aproximaţii de formă pentru o mulţime de puncte, de exemplu. Poate de asemenea să fie utilizată pentru a aproxima forme mult mai

complexe. De exemplu, înfăşurătoarea convexă a unui poligon, în plan, sau a unui poliedru în spaţiul tridimensional, este înfăşurătoarea convexă a vârfurilor lui**.**

Pe de altă parte, mulţi algoritmi calculează înfăşurătoarea convexă ca un stadiu iniţial în executarea lor sau pentru a filtra punctele nesemnificative. De exemplu, pentru a găsi dreptunghiul sau triunghiul minimal care închide o multime de puncte, este suficient să se calculeze mai întâi înfăşurătoarea convexă a punctelor, şi după aceea putem determina mai simplu dreptunghiul sau triunghiul minim care o cuprinde.

Avem nevoie mai întâi de câteva definiţii:

- ◊ **Convexitate**: O mulţime S este *convexă* dacă, fiind date oricare două puncte *P*, *Q* ∈ **S**, segmentul de dreaptă *PQ* se regăseşte în întregime în interiorul mulţimii iniţiale: $PQ \subseteq \mathbf{S}$.
- ◊ **Acoperirea convexă a unei mulţimi:** *Acoperirea convexă* a unei mulţimi **S** este intersecţia tuturor mulţimilor convexe care conţin mulţimea **S,** sau intuitiv, mulţimea convexă minimă care conţine mulţimea **S**. Urmând notarea făcută în cartea noastră, noi vom nota aceasta cu **Conv(S)**.
- ◊ **Înfăşurătoarea convexă a unei mulţimi (*Convex Hull*):** *Înfăşurătoarea convexă* unei mulţimi *S* este frontiera acoperirii sale convexe **Fr(Conv(S)).**

O definiţie empirică pentru înfăşurătoarea convexă a unei mulţimi de puncte în plan pleacă de la ideea imaginară de a „înfăşura” mulţimea de puncte cu o bandă elastică. Obţinem astfel o curbă simplă închisă cu proprietatea că este cel mai scurt drum care „înconjoară” mulţimea de puncte dată. Dacă mulţimea de puncte este finită, obţinem un poligon cu vârfurile printre punctele date. Interiorul acestuia, împreună cu laturile, este acoperirea convexă a mulţimii de puncte date.-

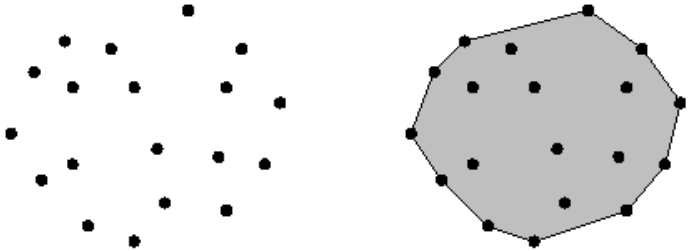


Fig. 7.1 O mulțime de puncte în plan, acoperirea lor convexă şi înfăşurătoarea convexă

Problema înfăşurătorii convexe (*CONVEXE HULL*): Fiind dată o mulțime de *n* puncte **P[i]** în plan, să se determine şi să se reprezinte grafic înfăşurătoarea convexă a mulțimii, şi deci, implicit, acoperirea sa convexă.

Există un algoritm simplu de complexitate O(*N*4) al determinării înfăşurătorii convexe a unei mulțimi finite de *N* puncte, care funcționează prin luarea în considerare a fiecărui punct din mulțime **P[i]** şi verificarea apartenenței sale la interiorul (sau laturile) unui triunghi format de trei alte puncte arbitrare distincte din mulțime **ΔP[j]P[k]P[h]**. Dacă această proprietate este verificată, punctul considerat este interior acoperirii convexe, şi nu poate aparține înfăşurătorii convexe, care este frontiera acesteia. Dacă nu există niciun triunghi format cu puncte din mulțime care să-l conțină pe **P[i]**, acesta este un punct de pe înfăşurătoarea convexă. Mai jos este prezentată o implementare în C a acestui algoritm. Pentru apartenența la interiorul unui triunghi, a se vedea şi subcapitolul 4.5. Funcția *cv[i]* întoarce *1* pentru apartenența punctului **P[i]** la înfăşurătoarea convexă, şi *0* pentru apartenența la interiorul acoperirii convexe.

```
void acoperire_convexa_naiva()
{
      int extrem[nmax];
      int i,j,k,h;
      for(i=0;i<n;i++){
            cv[i]=1;
            for(j=0;j<n;j++)
                  for(k=j+1;k<n;k++)
                        for(h=k+1;h<n;h++)
             if((i!=j)&&(i!=k)&&(i!=h)&&(punc
            t_in_triunghi(P[i],P[j],P[k],P[h
            ]) )) cv[i]=0;
            }
      }
```

Întrebarea este dacă putem să găsim un algoritm mai eficient.

7.2 Metoda împachetării (Wrapping)

Cel mai natural algoritm de determinare a acoperirii convexe, cu rădăcini în metodele empirice umane, este de a «înveli» (*wrapping*) mulțimea de puncte originală. Dacă se pleacă de la un punct aflat pe înfăşurătorea convexă (de exemplu, cel cu cea mai mică valoarea coordonatei Y) şi se trasează o semidreaptă mobilă ce pleacă de la orizontală şi se poate roti în sens trigonometric (de exemplu), primul punct intersectat de aceasta trebuie să fie pe înfăşurătoarea convexă. Apoi se retine acel punct şi se continuă procedura până când un alt punct este atins, etc., astfel până când "pachetul" este în întregime înfăşurat (punctul de început este inclus din nou). Diagrama următoare arata cum este descoperita înfăşurătoarea convexă în acest fel.

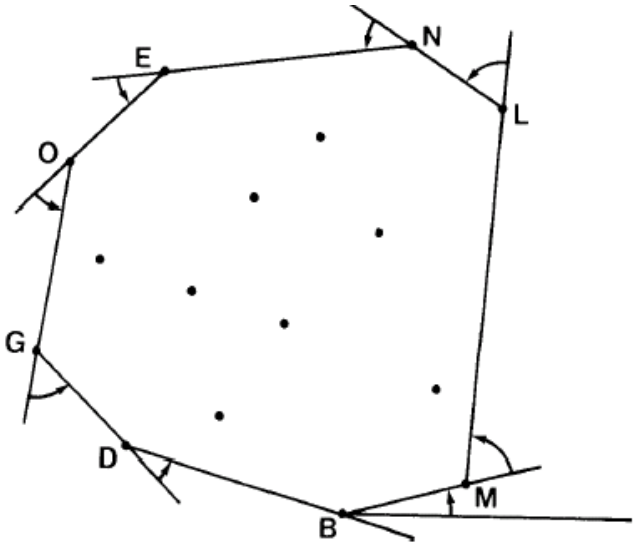


Fig. 7.2 Metoda împachetării pentru determinarea înfăşurătorii convexe

Desigur, nu trecem prin toate unghiurile posibile, facem doar un calcul standard de găsire a minimului unghiului unui vector față de orizontală, pentru a selecta următorul punct. Aceasta metodă este implementată uşor folosind funcția `theta(punct P1,P2:point)` descrisă în capitolul precedent (vezi subcap. 4.4), care poate fi gândită ca returnând unghiul intre P1, P2 şi orizontală (deşi acum returnează un număr calculat mult mai uşor cu aceleaşi proprietăți de ordine).

Următorul program găseşte înfăşurătoarea convexă a unui tablou de puncte p[1..N], reprezentate ca în descrierea din capitolul precedent (poziția p[N+1] din tablou este de asemenea utilizată, din considerente de ciclicitate). Inițializarea împachetării se face din punctul cel mai de jos din stănga (dacăă există mai multe puncte cu cea mai mică ordonată se alege cel cu cea mai mucă abcisă), folosind funcția

```
int jos_stanga(){
      int imin=0,i;
      for(i=0;i<n;i++)
{
      if((p[i].y<p[imin].y)||
((p[i].y==p[imin].y)&&(p[i].x<p[imin].x)))
            imin=i;
            }
      return imin;
      }
```

Unghiurile a doi vectori se determină cu ajutorul produsului scalar

```
float produs_scalar(vector u,vector v){
            return (u.x*v.x+u.y*v.y);
            }

float cos_unghi(vector u, vector v)
{
     return (produs_scalar(u,v)/
      (sqrt((produs_scalar(u,u))*(produs_sca
      lar(v,v)))));
     }
```

Procedura de împachetare arată astfel

```
int wrap(punct P[])
{int pc,pu; //punct curent, punct urmator
float cos_max;
vector dc,v; // vc- vectorul orizontal
dc.x=1;
dc.y=0;
      int imin=jos_stanga(); //initializare
      pc=imin;
      setcolor(GREEN);
      moveto(p[pc].x,p[pc].y);
      m=1; //punctul de plecare
```

```
do{
      cos_max=-1;
      for(int j=0;j<n;j++)
            if(j!=pc)
            {     v.x=p[j].x-p[pc].x;
                  v.y=p[j].y-p[pc].y;
                  float u=cos_unghi(v,dc);
                  if(u>cos_max){
                        cos_max=u;
                        pu=j;
                        }
                     else
                  if((u==cos_max)&&
                  (intre(p[pc],p[pu],p[j]))){
                     cos_max=u;
                     m++;
                     pu=j;
                     }
            }
            getch();
    // desenare linie
            lineto(p[pu].x,p[pu].y);
            dc.x=p[pu].x-p[pc].x;
            dc.y=p[pu].y-p[pc].y;
            pc=pu;
            m++;
      }while(pc!=imin);
}
```

La început punctul cu cea mai mică ordonată ***y*** este găsit si identificat prin indicele *imin* ca să oprească ciclul în buclă. Variabila *m* este mentinută ca *număr de puncte* incluse până acum pe înfăşurătoare, ***v*** este vectorul p[pc]p[j] dintre punctul curent (punctul până unde a ajuns înfăşuratoarea) şi un alt vârf a poligonului p[j], iar `u=cos_unghi(v,dc)` este cosinusul unghiului cu orizontala al vectorului **v**.

Cazul special în care se determină mai multe puncte coliniare pe înfăşurătoarea convexă este determinat de existența mai multor puncte care maximizează unghiul **u**. În acest caz, dacă ultimul

punct determinat este mai îndepărtat de cel precedent față de punctul curent al înfășurătorii, el este adăugat de asemenea infăşurătorii convexe. Determinarea ordinii a trei puncte coliniare se face folosind funcțiiile (definite în Capitolul 5)

```
int coliniare(punct A,punct B,punct C){
      if(orientare(A,B,C)!=0) return 0;
            else return 1;
}

int intre(punct A,punct B,punct C){
      if(!coliniare(A,B,C))  return 0;
      if(A.x!=C.x)
         return (((A.x<=B.x)&&(B.x<=C.x))||
            ((A.x>=B.x)&&(B.x>=C.x)));
      else
          return  (((A.y<=B.y)&&(B.y<=C.y))||
         ((A.y>=B.y)&&(B.y>=C.y)));
            }
```

Programul este aproape similar cu sortarea prin selectie, în care succesiv alegem cel mai bun punct neales inca , folosind o metoda bruta de cautare a minimului. Dezavantajul major al metodei este în cel mai rau caz, cand toate punctele cad pe infasurarea convexa , timpul de executie este proportional cu N^2.

Un aspect placut al acestei metode este ca se generalizează la trei (sau mai multe) dimensiuni. Înfăşurarea convexa a unui set de puncte într-un spațiu tridimensional este un obiect tridimensional cu fețe plate.

7.3 Scanarea Graham

Urmatoarea metoda pe care o vom examina, inventata de R. L. Graham în 1972, este interesantă pentr că cele mai multe din calculele implicate sunt dedicate unei simple ordonări: algoritmul include o sortare urmata de un calcul necostisitor (deşi nu evident).

Algoritmul porneste cu construirea unei linii poligonale simple trecând prin punctele date, utilizând metoda din Capitolul 5: Se obține astfel poligonul închis simplu din sectiunea 5.4. Se observă că p[N], p[1] şi p[2] sunt puncte consecutive pe înfăşurîtoare; în principal am rulat prima iterație a procedurii de înfăşurare prin împachetare.

Calculul infasurarii convexe este completat prin repetare, incercand sa punem fiecare punct pe infasurare şi eliminand punctele deja puse care nu pot fi pe aceasta. Pentru exemplu nostru, consideram punctele în ordinea B M J L N P K F I E C 0 A H G D.

Testul prin care stabilim care puncte sunt eliminate nu este dificil. După ce fiecare punct a fost adăugat, vom presupune că am eliminat suficiente puncte, astfel încât ceea ce am descoperit până acum ar putea fi parte a convex hull, pe baza punctelor pe care le deținem până acum. Algoritmul se bazează pe faptul că toate punctele din setul de puncte trebuie să se situeze pe aceeaşi parte a fiecărei margini a părții exterioare convexe. De fiecare dată când vom lua în considerare un punct, vom elimina din exterior orice margine care încalcă această condiție.

Testul pentru eliminarea punctelor este următorul: când examinăm un nou punct p[i], eliminăm p[k] din partea exterioară dacă linia dintre p[k] şi p[k-1] se regăseşte între p[i] şi p[1]. Dacă p[i] şi p[1] sunt pe aceeaşi parte a liniei, atunci p[k] ar putea fi pe exterior, deci nu îl eliminăm. Diagrama următoare arată situația exemplului nostru când L e considerat.

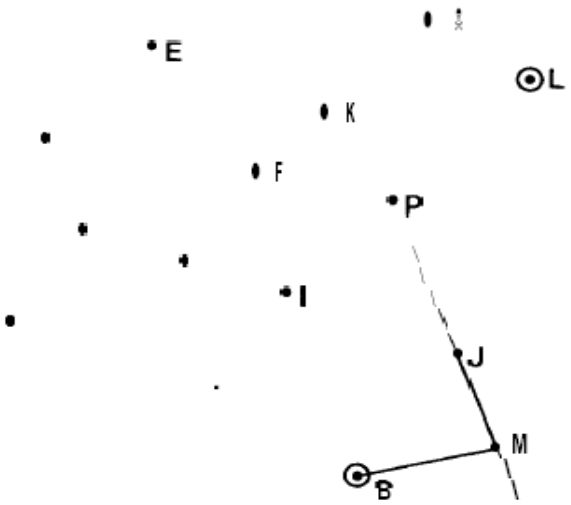


Fig. 7.3 Primii paşi la scanarea Graham

Linia extinsă JM este între L şi B, deci nu poate să fie pe în exterior. Acum L, N şi P sunt adăugate părții exterioare, apoi P este eliminat când este luat în considerare K (pentru că linia extinsă NP este între B şi K), atunci F şi I sunt adăugate, ducând la situația în care este luat în considerare E.

În aceată etapă, I trebuie să fie eliminat pentru că FI se regăseşte între E şi B, apoi F şi K trebuie să fie eliminate pentru că NFK se află între E şi B. Continuând în acest mod, vom ajunge înapoi la B, după cum este ilustrat mai jos.

Liniile punctate din diagramă sunt muchiile care au fost incluse, apoi eliminate. Sortarea inițiala garantează că fiecare punct este considerat ca un posibil punct exterior, deoarece toate punctele considerate mai devreme au o valoare *theta* mai mică. Fiecare linie care supraviețuieşte eliminării are proprietatea că fiecare punct se află pe aceeaşi parte ca şi p[1], ceea ce implică că trebuie să se regăsească pe partea exterioară.

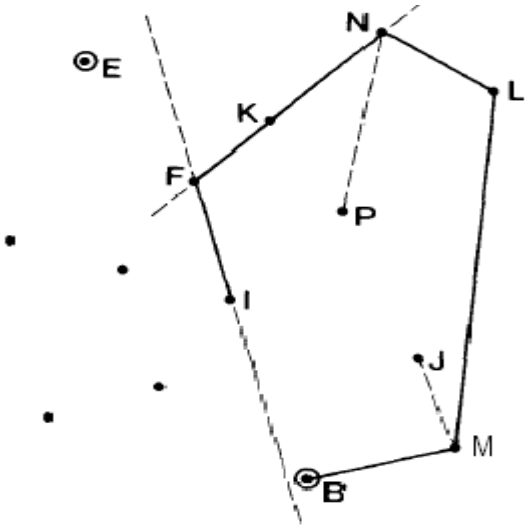


Fig. 7.4 Punctul I se elimină

Odată ce metoda este înțeleasă, implementarea este uşoară. În primul rând, punctul cu cea mai mica valoare y este schimbat cu p[0] (în cazul în care sunt mai multe puncte cu acelaşi *z* minim, se consideră cel cu *x* minim: cel mai de jos-stânga punct). Următorul program generează acoperirea convexă a setului de puncte p[0,...,N-1], folosind o stivă *s*[] pentru a păstra vârfurile înfăşurătorii convexe.

```
int jos_stanga(){
      int imin=0,i;
      for(i=0;i<n;i++){
      if((p[i].y<p[imin].y)||((p[i].y==p[imin].y
           )&&(p[i].x<p[imin].x)))
                  imin=i;
                  }
      return imin;
```

```
}

int mai_mare(int i,int j) {//i>j
          int imin=jos_stanga();
          int aux=orientare(p[imin],p[j],p[i]);
          if ((aux>0)||((aux==0)
                  &&intre(p[imin],p[j],p[i])))
            return 1;
          else return 0;
         }

void interschimba(int i,int j){
         punct aux;
           aux=p[i];
           p[i]=p[j];
           p[j]=aux;
          }

void Graham(){
       int imin=jos_stanga();
       int s[100];
       int t=1,i,j;
       interschimba(0,imin);
      //ordonarea punctelor
       for(i=1;i<n;i++)
         for(j=i+1;j<n;j++)
          if(mai_mare(i,j))   interschimba(i,j);

       s[0]=0;s[1]=1;// initializarea stivei
       i=2;
       while(i<n){
       if(orientare(p[s[t-1]],p[s[t]],p[i])>0)
          {t++;  // adaugare punct in stivă
            s[t]=i;
            i++;
            }
          else t--; // eliminare punct din stivă
      }
```

unde *mai_mare(i,j)* întoarce o valoare pozitivă dacă punctele `p[imin], p[j]` şi `p[i]` sunt orientate pozitiv sau sunt coliniare, cu `p[j]` între celelelte două.

Bucla *while* menține o acoperire parțială în `s[0],..., s[t]` (stiva conține indicii punctelordin înfăşurătoarea convexă). Pentru fiecare nouă valoare *i* considerată, dacă `orientare(p[s[t-1]],p[s[t]],p[i])>0, t` este incrementat şi un nou punct este adăugat în stivă, altfel *t* este deincrementat pentru a elimina puncte din acoperirea parțială. Următorul tabel arată conținutul mulțimii p de fiecare dată când un nou punct este luat în considerare în exemplul de mai sus.

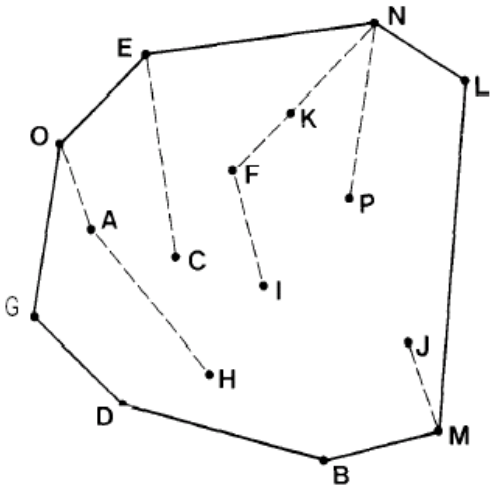
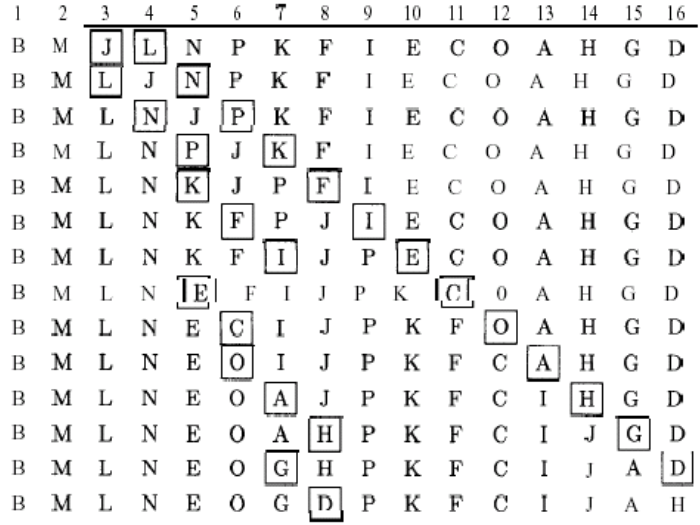


Fig. 7.5 Scanarea Graham completă

Acest tabel descrie, pentru i de la 4 la N, soluția scanării Graham când p[i] este luat primul în considerare, cu p[M]si p[i] încadrate.

Programul de mai sus poate da greş daca există mai mult de un punct cu cea mai joasa coordonata y, mai puțin daca *theta* e

modificat pentru a sorta corespunzator punctele colineare, cum e descris la subcapitolul precedent. În mod alternativ, calculul minimului poate fi modificat pentru a găsi punctul care are cea mai mică valoare a coordonatei x dintre toate punctele cu cea mai joasă coordonată y.



Un motiv pentru care această metodă este interesantă îl constituie faptul că ea reprezintă o formă simplă de *backtracking*.

7.4 Metoda rapidă prin selecție (*QuickHull*)

Aproape orice metodă de determinare a acoperirii convexe poate fi imbunatățită printr-o idee dezvoltată independent de către **W.F.Eddy şi R.Floyd**. Ideea generala e simplă: se aleg 4 puncte care se află sigur pe înfășurătoarea convexă, apoi se „videază” spațiul din interiorul patrulaterului format de aceste patru puncte. Aceasta lasă în mod semnificativ mai puține puncte pentru a fi luate în considerare cu scanarea Graham sau cu tehnica de împachetare.

Eventual, meoda selecției celor patru puncte şi a eliminării interiorului patrulaterului format de ele poate fi utilizată recursiv, obținând astfel algoritmul ***QuickHull.***

Cele 4 puncte cunoscute ca fiind sugur în înfățurătoare ar trebui alese luând în considerare informațiile despre acestea. În absența oricăror informații, cele mai simplu de folosit puncte sunt acelea cu cea mai mica şi cea mai mare coordonată x şi coordonată y. (Totuşi ar fi mai bine să adaptăm alegerea punctelor la distribuția datelor pe care le introducem. De exemplu, dacă toate valorile extreme x şi y sunt egale -o distribuție rectangulară- atunci alegerea punctelor prin scanarea dinspre colturi ar fi o metodă mai bună: se determină cele patru puncte cu cea mai mare şi mica sumă şi diferență dintre cele doua coordonate). Diagrama de mai jos arata că doar A şi J supraviețuiesc aplicării acestei tehnici în exemplul nostru de puncte.

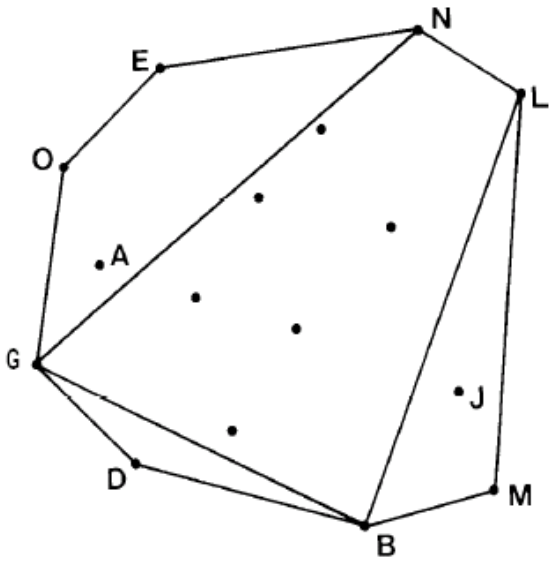


Fig. 7.6 Selecția rapidă cu eliminarea punctelor interioare

Versiunea recursivă a acestei tehnici e foarte similară procedurii *Quicksort* pentru selecție. La fel ca şi acestă procedură, are o complexitate O(N2), în cel mai rău caz.. De exemplu, dacă toate punctele originale se află pe înfăşurătoarea convexă parțială, atunci nici un punct nu va fi exclus la pasul recursiv.

7.5. Teme de laborator

1. Implementați algoritmul Graham descris la capitolul 7.3.

2. Se consideră următoarea implementare a algoritmului de împachetare care foloseşte funcția cosinus definită cu ajutorul produsului scalar. Modificați această implementare pentru a utiliza metoda descrisă în capitolul 7.2

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
#include<stdlib.h>
using namespace std;

typedef struct{
      float x,y;
      }punct;


int n; //numarul de puncte
      punct p[100];

typedef struct{
      float x,y;
}vector;


float produs_scalar(vector u,vector v){
            return (u.x*v.x+u.y*v.y);
            }
float cos_unghi(vector u, vector v){
```

```
     return
(produs_scalar(u,v)/(sqrt((produs_scalar(u,u))*(pr
odus_scalar(v,v)))));
     }

int jos_stanga(){
      int imin=0,i;
      for(i=0;i<n;i++){

      if((p[i].y<p[imin].y)||((p[i].y==p[imin].y)&
&(p[i].x<p[imin].x)))
            imin=i;
            }
      return imin;
      }


int orientare(punct A,punct B,punct C){
            float d=(float)(B.x-A.x)*(C.y-A.y)-
(float)(B.y-A.y)*(C.x-A.x);
      if (d>0) return 1;
        if (d<0) return -1;
      if (d==0) return 0;
}

int coliniare(punct A,punct B,punct C){
      if(orientare(A,B,C)!=0) return 0;
            else return 1;
}

int intre(punct A,punct B,punct C){
            if(!coliniare(A,B,C))  return 0;
            if(A.x!=C.x)      return
(((A.x<=B.x)&&(B.x<=C.x))||((A.x>=B.x)&&(B.x>=C.x)
));
                  else      return
      (((A.y<=B.y)&&(B.y<=C.y))||((A.y>=B.y)&&(B.y
>=C.y)));
      }


int punct_in_triunghi(punct M,punct A,punct
B,punct C){
            if (orientare(A,B,C)>0) return
((orientare(M,A,B)>=0) && (orientare(M,B,C)>=0)
&& (orientare(M,C,A)>=0));
```

```
      if(orientare(A,B,C)<0) return
((orientare(M,A,B)<=0)&&(orientare(M,B,C)<=0)&&

      (orientare(M,C,A)<=0));
}


float aria(punct A,punct B,punct C){
            float d=(float)(B.x-A.x)*(C.y-A.y)-
(float)(B.y-A.y)*(C.x-A.x);
            return fabs(d/2.0);
            }


int dreapta_jos(){
             int imin=0,i;
  for(i=0; i<n; i++)
  if
((p[i].x>p[imin].x)||((p[i].x==p[imin].x)&&(p[i].y
<p[imin].y)))
      imin=i;
      return imin;
}



int stanga_sus(){
   int imin=0,i;
   for(i=0; i<n; i++)
 if
((p[i].x<p[imin].x)||((p[i].x==p[imin].x)&&(p[i].y
>p[imin].y)))
      imin=i;
      return imin;
            }

void initializare(){
printf("n=");
scanf("%d",&n);
//PLEACA DE FIECARE DATA DIN ALT PUNCT
for(int i=0;i<n;i++){
 p[i].x=rand()%400;
             p[i].y=rand()%400;
}
}
```

```
void desenare(){
            char c[3];
            int i;
            setcolor(LIGHTBLUE);
            outtextxy(3,5,"Infãsurãtoarea
convexa");

      outtextxy(3,20,"_____________________________
__________");
            for(i=0;i<n;i++){
                  itoa(i,c,10);
                  setcolor(2);
                  circle(p[i].x,480-p[i].y,2);
                  setcolor(RED);
                  outtextxy(p[i].x,480-p[i].y,c);
                  }
            }


void impachetare(){
      int pc,pu;
      float cos_max;
      vector dc,v;
      dc.x=1;
      dc.y=0;
      int imin=jos_stanga();
      pc=imin;
      setcolor(GREEN);
      moveto(p[pc].x,480-p[pc].y);
      do{
            cos_max=-1;
            for(int j=0;j<n;j++)
                  if(j!=pc){
                        v.x=p[j].x-p[pc].x;
                        v.y=p[j].y-p[pc].y;
                        float u=cos_unghi(v,dc);
                        if(u>cos_max){
                              cos_max=u;
                              pu=j;
                              }
                        else

      if((u==cos_max)&&(intre(p[pc],p[pu],p[j]))){
                                    cos_max=u;
                                    pu=j;
                                    }
                              }
```

```
            getch();
            lineto(p[pu].x,480-p[pu].y);
            dc.x=p[pu].x-p[pc].x;
            dc.y=p[pu].y-p[pc].y;
            pc=pu;
      }while(pc!=imin);
}

void afisare(){
    for(int i=0;i<n;i++)
      cout<<"Punctul "<<i<<"  are
coord:"<<p[i].x<<",\t"<<p[i].y<<"\n";
     }

int  main(){
initializare();
//int gd=DETECT,gm;
//clrscr();
initwindow(600,500);

setbkcolor(15);
cleardevice();
desenare();
afisare();
impachetare();

getch();
closegraph();
      }
```

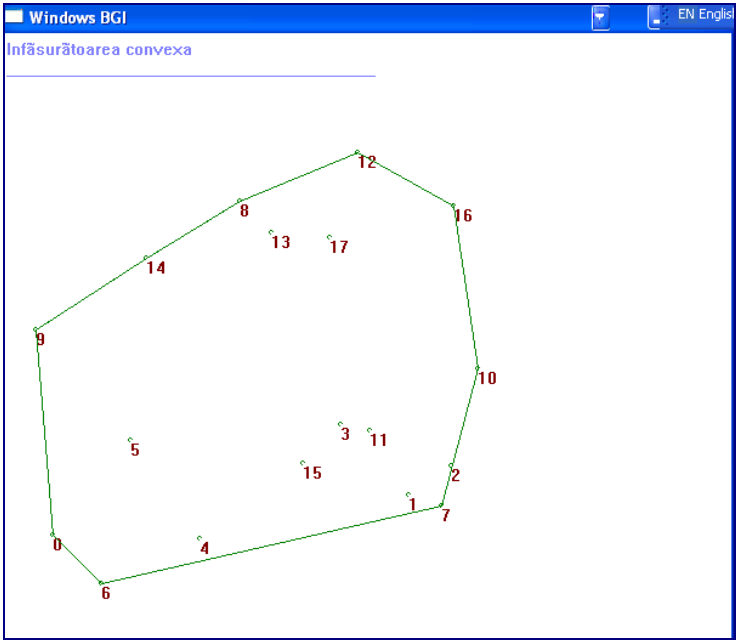


Fig. 7.6 Ecranul de execuție al algoritmului descris mai sus

3. Implementați algoritmul QuickHull descris la capitolul 7.4.

Capitolul VII. Subdomenii ale unui poligon

Descriere Generală

În acest capitol am abordat două dintre problemele cele mai cunoscute ale detereminării unor subdomenii ale unui poligon: problema nucleului şi problema partiției poligonului în triunghiuri.

Obiective

- Descrirea unui algoritm de determinare a nucleului unui poligon
- Introducere în problematica triangulărilor.
- Prezentarea triangulării unui poligon arbitrar

8.1. Problema nucleului

Printre problemele geometrice de calcul se numără şi problema determinării nucleului poligonului simplu. Nucleul unui poligon simplu P este, în general, format din mulțimea de puncte ale poligonului astfel încât oricare dintre ele, fiind unit cu orice alt punct al poligonului, formează un segment ce aparține în întregime poligonului (fig. 8.1).

Nu întotdeauna un poligon simplu are nucleu nevid (fig. 8.2).

Nucleul poligonului (dacă el există) este şi el un poligon, dar, spre deosebire de poligonul inițial, este întotdeauna convex.

Problema determinării nucleului unui poligon are o formă cunoscută sub denumirea de „Problema (simplă) a galeriei de artă”:

Se consideră o galerie de artă deformă poligonală dată. Să se determine, dacă există, zona

interioară din care un observator poate vedea toți pereții galeriei, fără a se deplasa.

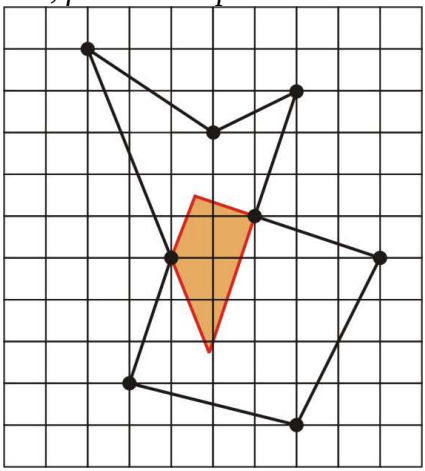


Fig. 8.1 Nucleul unui poligon (în zona centrală)

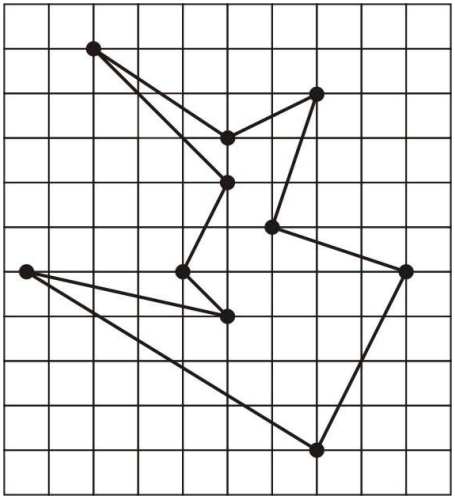


Fig. 8.2 Poligon fără nucleu

Date inițiale. Fie un poligon simplu P cu N vârfuri. Se consideră că poligonul este descris prin lista vârfurilor sale p[i] i=1,..N, parcurse în direcția mișcării acelor de ceasornic. Fiecare vârf p[i] este descris de coordonatele sale (x[i] , y[j]).

Algoritmul de determinare a nucleului necesită o construcție secundară: un poligon convex Q care, inițial, conține poligonul P. Pentru determinarea poligonului Q pot fi abordate două metode:

a) construirea înfășurătoarei convexe pentru P;

b) construirea unui dreptunghi care conține P.

Metoda a doua este mult mai simplă. Ea se reduce la determinarea, pentru coordonatele vârfurilor P, a valorilor minime şi maxime ale abscisei şi ale ordonatei. În calitate de *Q* poate fi considerat dreptunghiul cu vârfurile având coordonatele *x* repectiv *y* extreme.

După construirea poligonului *Q* poate fi determinat nemijlocit nucleul. Metoda (sau cel puțin descrierea ei în limbajul uman) este foarte simplă:

Se analizează consecutiv laturile poligonului *P*, fiind parcurse în direcția mișcării acelor de ceasornic. Latura curentă *p[i]p[i*+1*]* se cercetează ca o dreaptă *l*. Se determină partea poligonului *Q* care se află în stânga vectorului *p[i]p[i*+1*]* şi se exclude din *Q*. Dacă poligonul *Q* este situat integral în stânga vectorului *p[i]p[i*+1*]*, cercetarea ia sfârşit – poligonul *P* nu are nucleu.

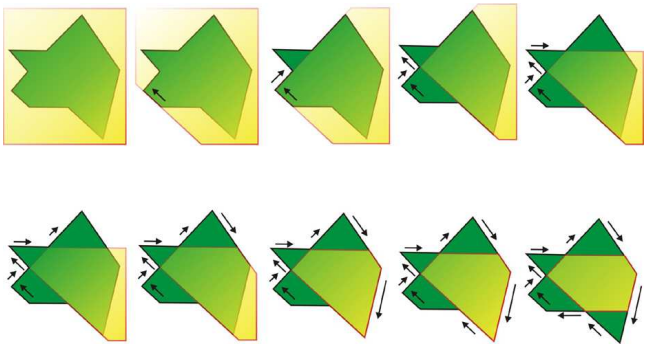


Fig. 8.3 Detreminarea nucleului poligonului P

Ca implementare concretă, pentru fiecare dreaptă *p[i]p[i+1]* se include intersecțiile cu laturile poligonului P ale acesteia în Q şi se exclud toate punctele din Q care au *Orientare*(*p[i],p[i+1], Q_k*) >0.

Procedeul se repetă până nu sunt analizate toate laturile poligonului *P*. În final, *Q* conține nucleul lui *P*.

Complexitatea algoritmului este $O(N^2)$. Se prelucrează *N* laturi ale poligonului *P*. Pentru fiecare latură se verifică intersecția cu maximum *N* laturi ale poligonului *Q*. Fiecare intersecție este determinată de un număr de operații mărginit de o constantă.

8.2. Triangularea poligoanelor

Triangularea unui poligon simplu este o operație folosită în diverse aplicații în care formele complexe urmează să fie descompuse într-un set de forme mai simple separate. Există multe domenii în care formele triunghiurilor sunt o problemă importantă (de exemplu triunghiurile obtuze ar trebui evitate in domeniul construcțiilor) dar există la fel de multe cazuri în care forma triunghiului nu este importantă. Noi vom lua în considerație problema în care se dă arbitrar un poligon simplu şi trebuie determinată o împărțire în triunghiuri (triangulație) arbitrară a acestuia.

Algoritmii de triangulare se ocupă cu determinarea unei mulțimi de triunghiuri având dată o mulțime de puncte 2D drept vârfuri. În general, am putea numi triangulare orice mulțime de triunghiuri în plan. Însă, dinmotive atât teoretice dar şi practice, o mulțime de triunghiuri definește o ***triangulare*** dacă:

1. nici un triunghi al unei triangulări nu este degenerat;
2. interioarele oricăror două triunghiuri nu se intersectează;
3. frontierele a două triunghiuri se pot intersecta numai după o latură comună sau un vârf comun;
4. reuniunea tuturor triughiurilor unei triangulări este egală cu domeniul triangulat;

Cazul cel mai simplu de la care putem începe studiul triangulărilor îl reprezintă triangularea *poligoanelor simple*. Un poligon simplu, prin defniție, este acel poligon care nu conține goluri şi ale cărui laturi nu se intersectează. Prima abordare într-un algoritm de triangulare a fost *decuparea colțurilor*, care a fost utilă prin simplitatea ei, dar având mai mult rol didactic,fiind un punct de pornire în studiul algoritmilor de triangulare.

Cazul particular al poligoanelor convexe necesită un algoritm de triangulare în timp liniar. Astfel triangularea se obține prin trasarea diagonalelor dintr-un vârf la celelalte vîrfuri. *Diagonala* reprezintă un segment de dreaptă interior poligonului care uneşte două vârfuri neconsecutive ale acestuia. Dar, descompunerea poligonului inițial în subpoligoane convexe, nu este tocmai simplă, aşa că s-a recurs la divizarea lui în regiuni monotone.

Astfel, un poligon simplu se numeşte *monoton față de o dreaptă d* dacă pentru orice dreaptă *d0* perpendiculară pe *d* intersecția poligonului cu *d0* este un segment de dreaptă, un punct sau mulțimea vidă. Apoi, s-au construit algoritmi de triangulare pentru aceste cazuri particulare de poligoane.

Există de asemenea foarte mulți algoritmi de complexitate $O(n \log(n))$ simpli pentru această problemă care se cunosc de mulți ani. O problemă rămasă deschisă multă vreme a fost cea a existenței unui algoritm de complexitate liniară $O(n)$. Problema a fost rezolvată de Chazelle în 1991, dar algoritmul este surprinzător de complicat, nu ar putea să rivalizeze cu algoritmi asimptotic mai lenți de complexitate $O(n \log n)$.

Pe de altă parte, fiind dat un poligon arbitrar cu *n* vărfuri, avem în general o mulțime de triangulări posibile, iar criteriile de a departaja o anumită triangulare de o alta din punct de vedere valoric depind de scopul urmărit. Astfel putem avea triangulări cu număr minim de triunghiuri, cu suma tuturor muchiilor minimale sau verificând alte criterii de optim, Cea mai bună metodă de a gasi o triangulare optimă pentru o problemă dată ar fi de a genera toate triangulările posibile şi ale compara una cu alta; acesta nu este lucru

fezabil pentru un poligon cu mai multe varfuri. Cu cât numărul de vârfuri creşte, numărul triangulărilor posibile se măreşte.

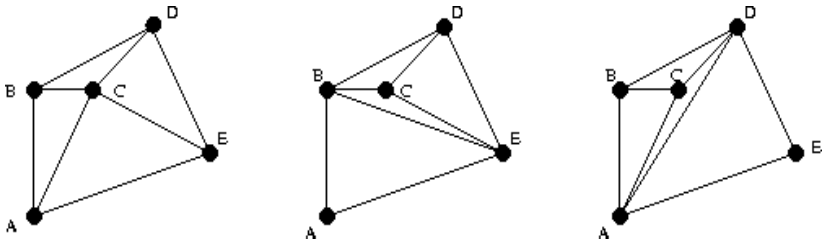


Figura 8.4. Trei triangulări posibile pentru un poligon cu un punct interior fixat

8.2.1 Triangularea poligoanelor convexe

Fie *P* un poligon convex, dat prin lista vârfurilor p[1], ... ,p[N] în ordinea parcurgerii lor. Pentru a construi o triangulare în *P*, este suficient să fie construite segmentele diagonale p[1]p[3],..., p[1]p[N-1]− (fig. 8.5).

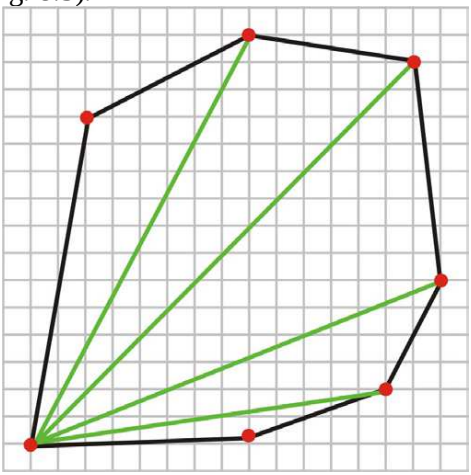


Figura 8.5.Triangularea unui poligon convex

Numărul diagonalelor necesare este egal cu *N-3* (proporţional cu numărul de vârfuri ale poligonului). Construirea unei diagonale necesită un număr constant de operaţii. Triangularea poligoanelor convexe poate fi utilă pentru calculul ariei lui. Aria poligonului este egală cu suma ariilor triunghiurilor din triangularizare.

În coctinuarea acestui capitol vom folosi o structură puţin mai complexă de definire a unui polinom, structură care conţine ca informaţii despre un vârf nu doar coordonatele sale, ci şi date despre poziţia relativă faţă de alte vărfuri. Mai jos am prezentat o metodă de definire, introducere şi afişare a unui poligon:

```
typedef struct {
      float x,y;
      int pred,succ;
      }punct;
typedef punct poligon[100];
poligon p;

void citire_poligon(){
      printf("n=");
      scanf("%d",&n);
      for (int i=1;i<=n;i++){
            printf("dati p[%d].x",i);
            scanf("%d",&p[i].x);
            printf("dati p[%d].y",i);
            scanf("%d",&p[i].y);
            };
      p[0].x=p[n].x; p[0].y= p[n].y
      // Poligonul este vazut ca o lista circulara
      for (int i=1;i<n=;i++){
            p[i].pred=(n+i-1)%n;
            p[i].succ=(i+1)%n;
            };
}

void desenare_poligon(int n, poligon p){
    char c[3];
```

```
     float x,y, x1,y1;
     int i,j;
      j=0; //punctul de plecare
      for (int i = 0; i < n; i++)
       { x=p[j].x;
         y=p[j].y;
         x1=p[ p[j].succ].x;
         y1=p[ p[j].succ].y;
           setcolor(10);
           itoa(i,c,10);
           outtextxy(x+5,y-5,c);
           setcolor(4);
           line(x,y,x1,y1);
           circle(x,y,2);
         j=p[j].succ;
      }
```

Din considerente de implementare în C, punctul `p[n]` a fost înlocuit cu `p[0]`; `p[i].pred` conține vârful precedent în parcurgerea directă a poligonului, iar `p[i].succ` conține vârful succesor. Instrucțiunea `j=p[j].succ;` asigură trecerea la vărful succesor după trasarea unei laturi.

Următoarea secvență verifică dacă un poligon este simplu (laturile sale nu se intersectează):

```
int poligon_simplu(int n,poligon p)
{ int i,k;
  for (i=0;i<n;i++)
    for (k=0;k<n;k++)
       if((i!=k)&&(i!=(k+1)%n)&&(k!=(i+1)%n)&&
    intersectie_segmente(p[i],p[(i+1)%n],p[k],
              p[(k+1)%n])) return 0;
return 1;
}
```

iar secvența de mai jos verifică dacă poligonul este convex

```
int poligon_convex(int n,poligon p)
{ int i,convex=1;
  for (i=1;i<n-2;i++)
```

```
       if(orientare(p[0],p[1],p[2])
            *orientare(p[i],p[i+1],p[(i+2)%n]))
            convex=0;
 return convex;
}
```

Determinarea unei triangulări a unui poligon convex se poate executa simplu prin

```
typedef struct {
      punct A,B,C;
      }triangulate;
triangulare T;
int nr_triunghiuri=0;

void triangulare_convexa(int n, poligon p)
{int i, nr_triunghi=0;
  for(i=1; i<n-1; i++)
      {nr_triunghi++;
       T[nr_triunghi].A=p[0];
       T[nr_triunghi].B=p[i];
       T[nr_triunghi].C=p[p[i].succ];}
}
```

Numărul de triunghiuri astfel obținut este egal cu *n-2*.

8.2.2 Triangularea poligoanelor simple arbitrare

În cazul poligoanelor simple ne-convexe nu este posibil să se aplice direct metoda din secțiunea precedentă, deoarece apar două condiții suplimentare care trebuie verificate:
1) aparține oare o anumită diagonală interioruluui poligonului, exteriorului poligonului sau intersectează anumite laturi;
2) triunghiurile care formează triangularizarea nu au neapărat toate un vârf comun..

Pentru a rezolva aceste probleme una din metodele cele mai simple constă în „decuparea urechilor”, metodă ce a fost inspirată de ideea de a transforma poligonul concav într-unul convex prin eliminarea anumitor vârfuri (cele plasate la exterior).

În descrierea formală se folosesc următoarele noțiuni:

Definiția 8.1

Un vârf p[i] al unui poligon simplu se spune ***componentă E*** *(parte a unei “urechi” –* ear *în engleză) dacă segmentul de dreaptă (p[i-1] p[i+1]) este o diagonală interioară a poligonului (se regăseşte complet în interiorul acestuia).*

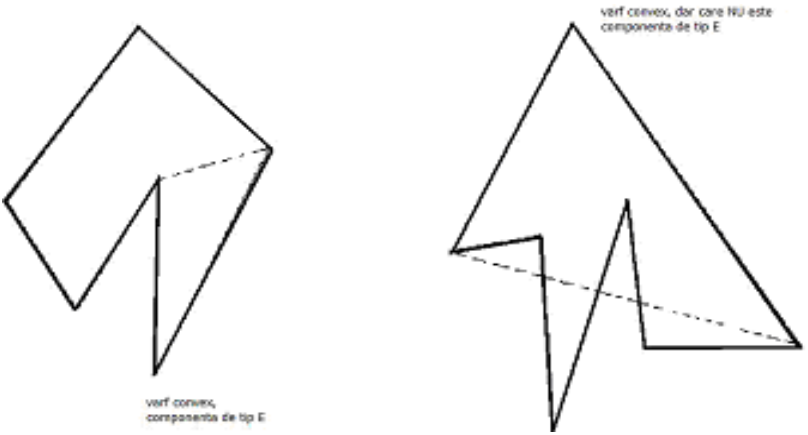


Figura 8.6.Un vârf- componentă E (la stânga), față cu un vârf care nu este componentă E,

Definiția 8.2

Un vârf p[i] al unui poligon simplu se spune ***componentă M*** *dacă segmentul de dreaptă (p[i-1] p[i+1]) este o diagonală exterioară a poligonului (se regăseşte complet în exteriorul acestuia).*

Un exemplu de vârf componentă M est edat în fig. 8.7

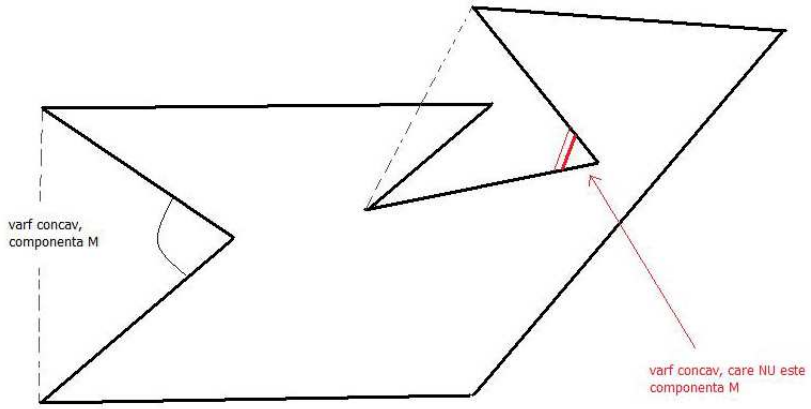


Figura 8.7.Un vârf- componentă M (la stânga), față cu un vârf care nu este componentă M,

De remarcat că orice componentă E a unui poligon poate defini un triunghi complet interior poligonului, deci un triunghi care poate fi inclus în triangulare (o „ureche") şi decupat din poligon. Noul poligon redus va avea cu un vărf mai puțin. Ca atare se poate determina un alt vârf care este componentă E a acestuia, iar procedura de tăere va continua pâna când mai rămân doar trei vârfuri (ultimul triunghi al triangularizării).

Testarea dacă diagonala *p[j]p[k]* este interioară verifică faptul că aceata nu are intersecții cu laturile care nu o definesc, iar mijlocul ei aparține interiorului poligonului

```
int diagonala_int(int j, int k){
      int i = p[j].succ;
      punct M;
      while (i != p[k].pred)
            if (intersectie_segmente(p[j],p[k],
                  p[i],p[p[i].succ]))
                  return 0;
            else  i = p[i].succ;
      i = p[k].succ;
```

```
      while (i != p[j].pred)
            if (intersectie_segmente(p[j],p[k],
            p[i],p[p[i].succ]))
                  return 0;
            else  i = p[i].succ;
      M.x=(p[j].x+p[k].x)/2;
      M.y=(p[j].y+p[k].y)/2;
      if(punct_in_poligon(M,p)) return 1;
        else
      return 0;
}
```

(Funcția `punct_in_poligon(M,p)` se construiește conform descrierii din subcapitolul 5.5) Determinarea triangulării devine

```
void triangulare_poligon(int n, poligon p)
{int i,j,k,d;
 Int nr_triunghi=0;
 i=0; //varful de plecare
 d=0; //nr de varfuri taiate
 while(d>=3)
     { j=p[i].pred;
       k=p[i].succ;
       if( diagonala_int(j,k) )
            {nr_triunghi++;
             T[nr_triunghi].A=p[j];
             T[nr_triunghi].B=p[i];
             T[nr_triunghi].C=p[k];
             p[j].succ=k; //”taie” varful i
             i=j;
            }
              else i=k;
     }
}
```

Exemplu

Se consideră poligonul din figura 8.8

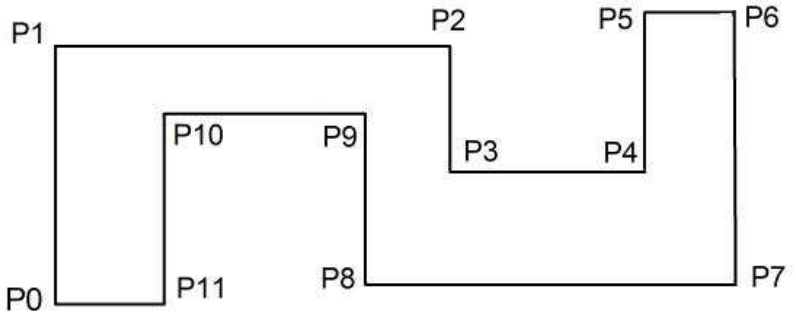


Figura 8.8.Un poligon simplu ne-convex

Inițial, algoritmul testează P1 pentru a determina dacă este sau nu o componentă E. Scanarea este avansată peste P2, P3, P4 şi P5 când P5 este determinat ca fiind o componentă E.

Vârful P5 este decupat şi atunci P4 este testat din nou pentru poligonul redus (fig 8.9), determinându-se că nu este o componentă E.

Următorul vărf testat este P6. Este găsit ca fiind componentă E şi decupat. Vărful p4 este testat iar şi de această dată este componentă E şi este decupat.Vârfurile rămase vor fi decupate în ordinea următoare: P7, P3, P8, P2, P9, P1.

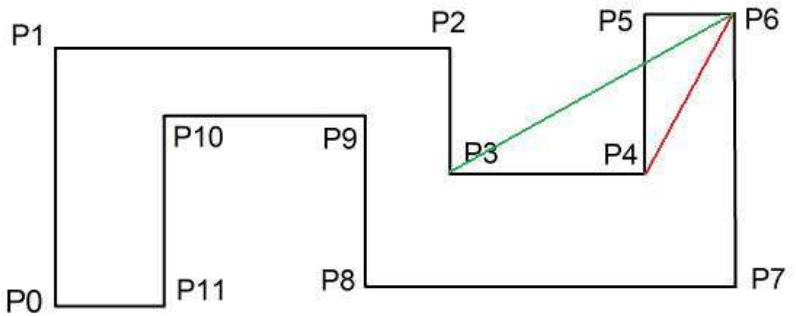


Figura 8.9. Decuparea unei urechide terminată de vârful P5. Segmentul (P3P6) determină o diagonală exterioară, deci P4 nu va fi decupat în această etapă

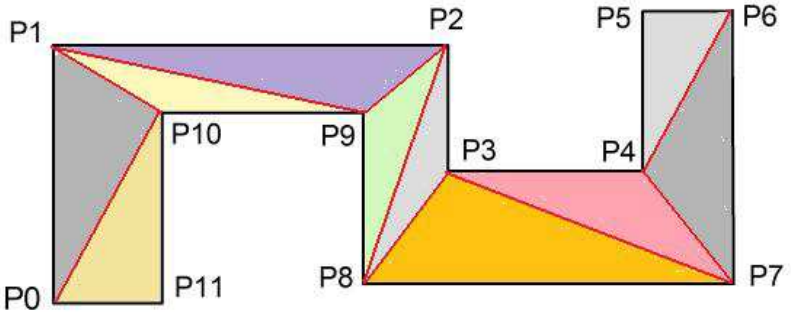


Figura 8.10.Triangularea finală.

La sfârşit se obține triangularea din figura 8.10

8.3. Teme de laborator

1. Implementați algoritmii descrişi la 8.1 şi 8.2.

2. Imaginați un algoritm care determină acoperirea convexă a unui poligon folosind „umplerea” urechilor. (Indicație: se folosesc diagonalele exterioare pentru a completa poligonul cu triunghiuri determinate de câte un vârf componentă M şi laturile sale adiacente)

Capitolul IX. Diagrame Voronoi şi triangulări Delaunay

Descriere Generală

În acest capitol am abordat două dintre problemele clasice ale geometriei computaționale. Problema trasării frontirerelor naturale între mai multe locații (a diagramelor Voronoi) şi problema determinării celei mai bune triangulări din punct de vedere al măsurătorilor topografice (triangularea Delaunay).

Obiective

- Descrirea unui algoritm de determinare a diagramelor Voronoi
- Introducere în problematica triangulărilor.
- Prezentarea triangulării Delaunay şi a legăturii acesteia cu diagramele Voronoi

9.1. Problema diagramelor Voronoi

Considerăm cazul de geografie sociala următor: într-o țară, un lanț de magazine deschide într-un număr de locații date câteva centre comerciale noi. Să se asigneze fiecare punct geografic din țară la o zonă care cuprinde un asemenea nou centru. Asignarea se va face dupa principul distanței minime până la centrul comercial. Echivalent, se cere determinarea tuturor "frontierelor" care separă zonele asociate fiecărui centru comercial, zone formate din punctele cele mai apropiate de unul sau altul dintre magazinele lanțului.
Fiecare astfel de zonă se numeşte regiune Voronoi. *Diagrama Voronoi* este formată din mulțimea punctelor care aparțin la cel puțin două regiuni Voronoi, sau, alfel spus, mulțimea frontierelor zonelor Voronoi.

Zonele Voronoi au proprietatea că sunt întotdeauna mulțimi convexe ([1], [7]). Pe de altă parte, considerând două centre arbitrare

P1 şi P2, este cunoscut din geometria elementară faptul că frontiera zonelor Voronoi asociate trebuie să cnțină cel puțin o parte a mediatoarei segmentului P1P2. În cazul în care avem trei centre comerciale necoliniare A,B,C, diagrama Voronoi asociată va cuprinde centrul O al cercului circumscris triunghiului ABC, împreună cu semidreptele mediatoare ale fiecărei laturi ce pleacă din O. Problema construcției diagramei Voronoi se complică însă când avem mai mult de trei centre.

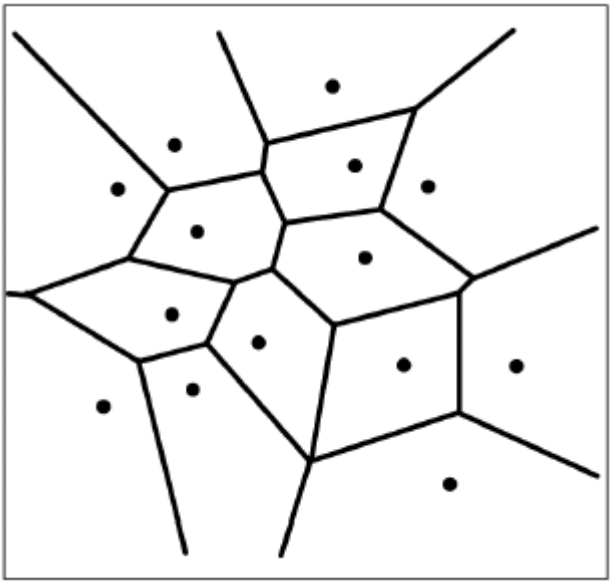


Figura 9.1 Exemplu de diagramă Voronoi

Ca aplicabilitate practică, diagramele Voronoi se folosesc în *geografia socială*, când dorim să determinăm aria de influență a unui factor asupra perimetrului înconjurător în 2D, de exemplu. Pentru fiecare locație vom găsi celula care cuprinde toate punctele cel mai apropiate de locația corespunzătoare, raportate la toate celelalte locații.

Diagramele Voronoi pot conduce de asemenea la *generarea fractalilor*. Pentru a crea un fractal, după ce am creat o diagramă

Voronoi din câteva puncte, vom adăuga apoi alte puncte şi creăm noi diagrame Voronoi în interiorul fiecăreia din cele de dinainte. Repetând procesul recursiv, putem genera chiar structura unei frunze.

9.2. Construcția algoritmică a diagramelor Voronoi

Considerăm în cele ce urmeauă, cu titlul de exemplu, o problemă a fost propusă la barajul pentru lotul olimpic din Timişoara în martie 1997 de Manuela Mateescu:

> *Postul de televiziune ProInfo are posibilitatea să-şi înființeze posturi locale în **n** (1<=n<=200) localități, ale căror coordonate, specificate relativ la un sistem de coordonate ortogonal cu centrul în coltul din stânga jos al unei hărți de dimensiuni LxH (1<=L,H<=30000), sunt cunoscute. Cum orice intenție trebuie realizată cu cheltuieli minime, se cere calculată cu precizie mai bună decat 0.01 puterea minima necesara pentru fiecare statie de emisie, astfel încăt sa acopere întreaga zonă arondată postului local corespunzator. Mai exact, zona arondată postului local P_i este formată din totalitatea punctelor din planul hărții care sunt mai apropiate de postul local P_i decât de orice alt post, iar puterea necesară stației de emisie este direct proportională cu suprafața zonei arondate postului, factorul de proportionalitate fiind egal cu 1 (unu) indiferent de condițiile de relief, climă, etc.*

Avem la intrare n puncte în planul XOY care sunt posturile locale P_i (1<=i<=n) si pentru care trebuie să determinăm pe rând "totalitatea punctelor din planul hărții care sunt mai apropiate de postul local P_i decat de orice alt post". Daca analizăm problema puțin, se observă că avem de determinat un anumit număr de poligoane. Aceste poligoane definesc Diagrama Voronoi.

Ideea rezolvarii problemei constă în determinarea punctelor egal depărtate între două posturi locare, unul fiind cel analizat P_i, iar celalat P_j. Aceste puncte aparţin frontierei zonelor Voronoi ale lui P_i respectiv P_j. Avem aşadar o latură comună celor doua poligoane-frontieră. Această latură comună este un segment de dreaptă pe mediatoarea segmentului (P_i,P_j).

Algoritmul direct de determinare a Diagamelor Voronoi parcurge fiecare punct de pe hartă care defineşte un post local P_i (1<=i<=n) şi îi atribuie iniţial ca poligon Voronoi chiar laturile care definesc harta, descrisă schematic de fereastra dreptunghiulară iniţială cu laturile (L,H).

Pasul urmator este de a determina toate mediatoarele laturilor (P_i,P_j) cu 1<=j<=n. Pentru fiecare mediatoare se calculează intersecţiile cu poligonul Voronoi curent şi se păstrează acea parte din poligonul Voronoi curent care este de aceeasi parte cu mediatoarea. Intersecţiile mediatoarei cu poligonul curent asociat lui P_i determina noi puncte-frontieră ale noului poligon Voronoi.

În final, dupa intersecţia cu toate mediatoarele se obtine zona Voronoi al postului P_i..

Pe de altă parte, am preferat în cele ce urmează prezentarea unei variantă de determinare a Diagramei Voronoi care utilizează noţiunea de Triangulare Delaunay.

9.3. Triangularea Delaunay a unui ansamblu de puncte

Tringularea Delaunay a unui ansamblu de puncte date S este definită de următoarea proprietate: orice cerc circumscris unuia dintre triunghiurile componente ale triangulării nu conţine nici un alt punct al ansamblului înâuntrul său. Aceată triangulare a fost introdusă în studiul cartografiei, pentru determinarea cu cele mai mici erori a ariilor unor suprafeţe. Principala ei proprietate se traduce prin reducerea la maxim a dimensiunii unghiurilor obtuze ale triunghiurilor interioare.

O triangulare Delaunay este în realitate structura duală în sens de graf asociată unei diagrame Voronoi. Principalele proprietăți ale unei tringulări Delaunay sunt [1]:

- Reuniunea triunghiurilor din triangulare reprezintă chiar acoperirea convexă a mulțimii S
- Fiecare triunghi aparținând triangulării corespunde unui vărf al diagramei Voronoi (vărful Voronoi corespunzător este centrul cercului circumscris triunghiului respectiv).
- Triunghiurile din triangularea Delaunay care au o latură situată pe frontiera acoperirii convexe a lui S corespund unei laturi nemărginite (semidrepte) a diagramei Voronoi.

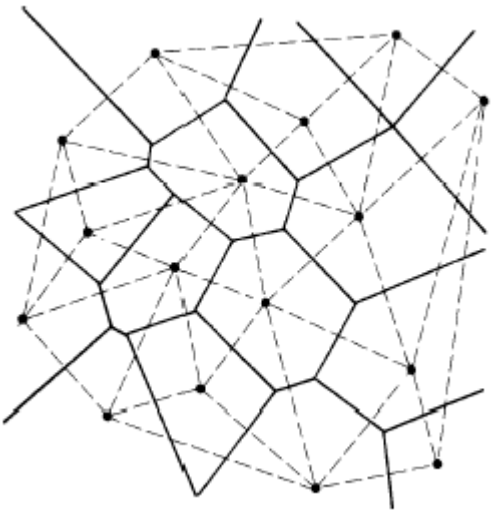


Figura 7.3 Triangularea Delaunay (punctată) asociată unei diagrame Voronoi (în linie continuă)

Proprietatea de definiție a triangulării Delaunay sugerează unul dintre algoritmii de de terminare a ei, descris în sub capitolul următor.

9.4. Algoritm de determinare a triangulării Delaunay şi a diagramei Voronoi

În continuare am prezentat un program complet, conceput în C, de determinare a diagramei Voronoi asociată unui ansamblu de n puncte generate aleator în plan $S=\{P_i\}$, bazat pe determinarea prealabilă a triangulării Delaunay asociată. Ideea centrală este de a verifica toate triunghiurile posibile de forma $P_iP_jP_k$ cu $1 \le i < j < k \le n$;I de a reține doar cele pentru care nici un alt punct al ansamblului S nu este conținut în interiorul cercului circumscris $\Delta P_iP_jP_k$. Algoritmul funcționează întrucât , în baza unei teoreme a lul Delaunay [5], triagularea Delaunay este unică.

Funcția `centru_cerc(punct A,punct B,punct C)` determină centrul cercului circumscris unui triunghi ABC, iar `punct_in_cerc(punct M,punct A,punct B,punct C)` determină apartenența la interiorul cercului circumscris determinat de punctele A,B,C.

Diagrama Voronoi se obține unind pur şi simplu centrele cercurilor circumscrise triunghiurilor Delaunay prin segmente de dreaptă care sunt obligatoriu perpendiculare pe laturile triangulației.

```
      //diagrama_     voronoi     si     triangulare
delaunay...........
      #include <stdio.h>
      #include <stdlib.h>
      #include <math.h>
      #include <graphics.h>
      #include <conio.h>
      #define nmax 40

      typedef struct{
        float x,y;
       }vector;
```

```
typedef struct{
  float x,y;
 }punct;

typedef struct{
  int v[4];
  }triunghi;

punct p[40];
int n,nt=0;
triunghi T[2*nmax];

void citire(){
int i;
 printf("n=");
 scanf("%d",&n);
 randomize();
 for (i=0;i<n;i++){
     p[i].x=100+random(300);
     p[i].y=100+random(300);
 }
}

void desenare(){
char c[3];
int i;
 for(i=0;i<n;i++){
  itoa(i,c,10);
  setcolor(YELLOW);
  circle(p[i].x,p[i].y,1);
  setcolor(RED);
  outtextxy(p[i].x+3,p[i].y+3,c);
 }
 getch();
}

void desenare_triunghi(punct A,punct B,punct
C,int culoare){
     setcolor(culoare);
     moveto(A.x,A.y);
     lineto(B.x,B.y);
     lineto(C.x,C.y);
     lineto(A.x,A.y);
     getch();
}
```

```
      float produs_scalar(vector u,vector v){
        return (u.x*v.x+u.y*v.y);
      }

      vector scade(punct A,punct B){
        vector v;
        v.x=A.x-B.x;
        v.y=A.y-B.y;
        return v;
      }

      float dist(punct a,punct b){
         return      sqrt((b.x-a.x)*(b.x-a.x)+(b.y-
a.y)*(b.y-a.y));
      }

      float tangenta(punct A,punct B,punct C){
        vector u=scade(A,B),v=scade(C,B);
        return
sqrt(produs_scalar(u,u)*produs_scalar(v,v)-
produs_scalar(u,v)*produs_scalar(u,v))/produs_scal
ar(u,v);
      }

      punct centru_cerc(punct A,punct B,punct C){
        vector v1=scade(B,A);
        vector v2=scade(C,B);
        vector v3=scade(A,C);
        punct o;
       if(produs_scalar(v1,v3)==0){
            o.x=(B.x+C.x)/2;
            o.y=(B.y+C.y)/2;
            return o;
       }
        if(produs_scalar(v1,v2)==0){
            o.x=(A.x+C.x)/2;
            o.y=(A.y+C.y)/2;
            return o;
       }
        if(produs_scalar(v2,v3)==0){
            o.x=(B.x+A.x)/2;
            o.y=(B.y+A.y)/2;
            return o;
       }
        float
ta=tangenta(B,A,C),tb=tangenta(A,B,C),tc=tangenta(
A,C,B);
```

```
        float suma=ta+tb+tc;

o.x=((tb+tc)*A.x+(ta+tc)*B.x+(ta+tb)*C.x)/(2*suma)
;

o.y=((tb+tc)*A.y+(ta+tc)*B.y+(ta+tb)*C.y)/(2*suma)
;
        return o;
      }

      void    semidreapta(punct    M,vector    v,int
culoare){
      float tmax=200,past=0.5;
      float t;
      for(t=0;t<=tmax;t+=past){
          putpixel(M.x+t*v.x,M.y+t*v.y,culoare);
          }
      }

      int semn(punct a,punct b,punct c)
       {
       double d1;
       d1=(double)(b.x-a.x)*(c.y-a.y)-
(double)(c.x-a.x)*(b.y-a.y);
       if (d1>0)return 1;
        else if(d1<0)return -1;
            else return 0;
       }


      void  mediatoare(punct  A,punct  B,punct  C,int
culoare){
      //deseneaza mediatoarea laturii BC
      vector v;
      if (semn(A,B,C)>0) {
         v.x=C.y-B.y;
         v.y=-(C.x-B.x);
            }
        else{
          v.x=-(C.y-B.y);
          v.y=C.x-B.x;
            }
       float t=sqrt(produs_scalar(v,v));
       v.x=v.x/t;v.y=v.y/t;
       punct O=centru_cerc(A,B,C);
       semidreapta(O,v,culoare);
      }
```

```
      void    mediatoare_triunghi(punct    a,punct
b,punct c,int culoare){
       mediatoare(a,b,c,culoare);
       mediatoare(b,c,a,culoare);
       mediatoare(c,a,b,culoare);
      }

      int  punct_in_triunghi(punct  M,punct  A,punct
B,punct C){
      if (semn(A,B,C)>0)
         return (semn(M,A,B)>=0&&
                        semn(M,B,C)>=0
                  &&semn(M,C,A)>=0);
        else  return (semn(M,A,B)<=0&&
                  semn(M,B,C)<=0
             &&semn(M,C,A)<=0);
      }

      int   punct_in_cerc(punct   M,punct   A,punct
B,punct C){
      punct O=centru_cerc(A,B,C);
      float raza=dist(O,A);
      if(dist(O,M)<=raza) return 1;
      else return 0;
      }

      void triangulare_delaunay(){
          int i,j,k,l,ok;
          for(i=0;i<n-2;i++)
             for(j=i+1;j<n-1;j++)
              for(k=j+1;k<n;k++){
                 ok=1;
                 l=0;
                 while(ok && l<n){
            if(l!=i&&l!=j&&l!=k&&punct_in_cerc(p[l
      ],p[i],p[j],p[k])) ok=0;
              l++;
                 }
                 if(ok){

desenare_triunghi(p[i],p[j],p[k],3);

//mediatoare_triunghi(p[i],p[j],p[k],4);
                 nt++;
                 T[nt].v[1]=i;
                 T[nt].v[2]=j;
```

```
                T[nt].v[3]=k;

              }
            }
      }

      void   verifica_latura(int   i0,int   j0,int
k0,int i,int culoare){
      //io,jo,ko sunt virfurile triunghiului i
      //functia analizeaza latura iojo
      int ok=0,j,ind,k;
      punct O1=centru_cerc(p[i0],p[j0],p[k0]);
      for(j=1;j<=nt;j++){
            ind=0;
            if(j!=i){
                  for(k=1;k<=3;k++){
                        if(T[j].v[k]==i0) ind++;
                        if(T[j].v[k]==j0) ind++;
                        }
                  if(ind==2){
                        punct
O2=centru_cerc(p[T[j].v[1]],p[T[j].v[2]],p[T[j].v[
3]]);
                        setcolor(culoare);
                        line(O1.x,O1.y,O2.x,O2.y);
                        ok=1;
                        }
                  }
            }
      if(ok==0)
mediatoare(p[k0],p[i0],p[j0],culoare);
      }

      void diagrama_voronoi(int culoare){
      int i,i0,j0,k0;
      for(i=1;i<=nt;i++){
            i0=T[i].v[1];j0=T[i].v[2];k0=T[i].v[3]
;
            verifica_latura(i0,j0,k0,i,culoare);

      verifica_latura(j0,k0,i0,i,culoare);
            verifica_latura(k0,i0,j0,i,culoare);
            }
      }

      void main(){
       citire();
```

```
 int gm, gd=DETECT;
 initgraph(&gd, &gm, "C:\\bc31\\bgi");
 desenare();
 triangulare_delaunay();
 diagrama_voronoi(RED) ;
getch();
}
```

9.6. Teme de laborator

1. Implementați algoritmul descris la 9.2.

2. Imaginați un algoritm care determină triangularea Delaunay cunoscând diagrama Voronoi a unui ansamblu de puncte.

Capitolul X. Problema celui mai scurt drum într-un câmp cu obstacole şi graful de vizibilitate

Descriere Generală

Problema drumului celui mai scurt printr-un câmp cu obstacole poligonale este formulată în acest capitol, fiind abordată prin prisma utilizării aşa-numitului „graf de vizibilitate”.

Obiective

- Descrirea unui algoritm de determinare a grafului de vizibilitate
- Introducere în problematica mişcării într-un câmp cu obstacole

10.1. Problema celui mai scurt drum într-un câmp cu obstacole

Fiind dat un set de *n* obstacole disjuncte poligonale în plan, şi două puncte **s** şi **t** care se află în afara de obstacole, **problema celui mai scurt drum** este de a stabili calea mai scurtă de la **s** la **t** care evită interioarele de obstacole. (Se poate trece de-a lungul marginilor sau prin colţurile obstacolelor.) Complementul interioarelor obstacolelor se numeşte *spaţiu liber*. Vrem să determinăm drumul cel mai scurt care trece în întregime prin spaţiu liber.

Soluţia acestei probleme se bazează pe aplicarea unei teoreme din [5] (pg. 131):

Teoremă

*Calea cea mai scurtă între două puncte **s** şi **t**, care evită un set de obstacole poligonale este o curba poligonală, ale cărei vârfuri sunt fie nodurile de obstacole, fie punctele terminale.*

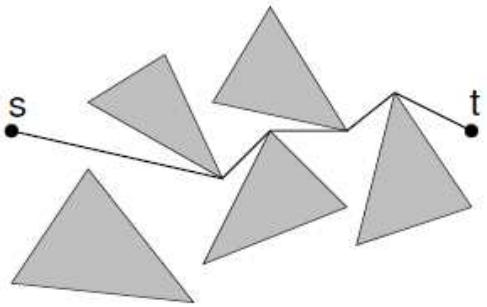


Fig 10.1. Cel mai scurt drum printr-un câmp cu obstacole

Din aceasta rezultă că marginile care definesc calea cea mai scurtă dintre **s** şi **t** sunt vârfuri de obstacole. Fiecare dintre aceste margini trebuie să aparţina obstacolelor iar fiecare segment care leagă două astfel de colţuri nu intersectează interiorul niciunui obstacol, ceea ce implică faptul că colţurile succesive din drumul ales trebuie să fie vizibile între ele. Mai formal, putem spune că două puncte P şi Q sunt reciproc vizibile dacă segmentul PQ nu intersectează interiorul niciunui obstacol.

Aceată proprietate permite introducerea noţiunii de „graf de vizibilitate".

10.2. Graful de vizibilitate

Definiţie 10.1

***Graful de vizibilitate** al unei mulţimi formată din două puncte S şi T şi un grup de obstacole poligonale în plan se defineşta ca graful format din cele două puncte împreună cu ansamblul vărfurilor obstacolelor, având ca muchii toate segmentele [PQ] formate din vârfuri vizibile între ele sau din muchii ale obsatcolelor .*

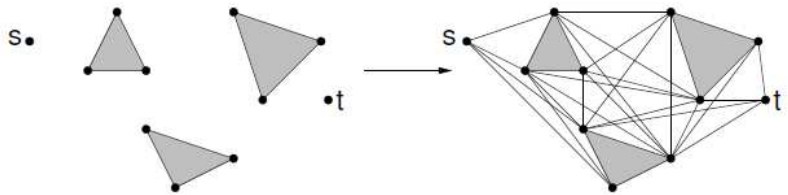


Fig 10.1. Graful de vizibilitate

Din definiția de mai sus rezultă că drumul cel mai scurt euclidian între S şi T poate fi calculat ca cel mai scurt drum în graful de vizibilitate, folosind de exemplu algoritmul Dijkstra. Lungimea muchiilor se determină evident folodind distanța euclidiană.

Singura problemă rămâne determinarea grafului de vizibilitate.

Dacă obstacolele sunt introduse sub forma unui vector de poligoane, este suficient să se verifice intersecțiile *proprii* dintre oricare segmente care leagă două vârfuri ale poligoanelor între ele sau cu punctele terminale cu oricare dintre laturile poligoanelor care formează obstacolele.

```
typedef struct {
      int x,y;
      }punct;
typedef punct poligon[20];
poligon p[20]; //numărul maxim de obstacole
int m,n[20],i,j;
punct S,T;

typedef struct {
      int ob,v; // ob=nr. Obstacolului,
                //v=nr. varfului
      }nod; // definire nod graf
// ob=0, v=0 -> punctual S
// ob=m+1, v=0 -> punctual T
nod ln[400]; //lista de varfuri a grafului
```

```
int graf[400][400];// matricea de adiacenta


void citire_obstacole()
{
      printf("Nr. De obstacole m=");
      scanf("%d",&m);
      for (int i=1;i<=m;i++)
      {printf("Cate varfuri are obstacolul %d?");
       scanf("%d",&n[i]);
       for (int j=1;j<=n[i];j++){
            printf("dati p[%d][%d].x",ij);
            scanf("%d",&p[i][j].x);
            printf("dati p[%d][%d].y",i,j);
            scanf("%d",&p[i][j].y);
            };
       }
}

void citire_pct_teminale()
{
      printf("Punctul de start x=");
      scanf("%d",&S.x);
      printf(" y=");
      scanf("%d",&S.y);
      printf("Punctul terminus x=");
      scanf("%d",&T.x);
      printf(" y=");
      scanf("%d",&T.y);
}

void graf_vizibilitate()
{ punct a,b,c,d;
  int i,j,k,h,nrnod=1;
  ln[0].ob=0; ln[0].v=0 //pct. de start
  for (i=1;i<=m;i++)
    for (j=1;i<=n[i];j++)
      { nrnod++;
        ln[nrnod].ob=i;
        ln[nrnod].v=j;
      };
```

```
  nrnod++;
  ln[nrnod].ob=m+1;
  ln[nrnod].v=0;  // pct. terminal
  for (i=0;i<=nrnod;i++)
    for (j=0;j<=nrnod;j++)
      graf[i][j]=1; //initializarea grafului
            //de vizibilitate, presupus complet

for (i=1;i<nrnod-1;i++)
    for (j=i;j<nrnod-;j++)
     for (k=1;k<nrnod-1;k++)
      for (h=k;h<nrnod;h++)
      {a=p[ln[i].ob][ln[i].v];
      b=p[ln[j].ob][ln[j].v];
      c=p[ln[k].ob][ln[k].v];
      d=p[ln[h].ob][ln[h].v];
       if(intersectie_proprie(a,b,c,d))
        graf[i][j]=0; //elimina muchie invizibila
            // intre varfurile obsatcolelor
      }


for (i=1;i<nrnod-1;i++)
     for (k=1;k<nrnod-1;k++)
      for (h=k;h<nrnod;h++)
      {a=p[ln[i].ob][ln[i].v];
       b=S;
       c=p[ln[k].ob][ln[k].v];
       d=p[ln[h].ob][ln[h].v];

       if(intersectie_proprie(a,b,c,d))
        graf[i][j]=0; //elimina muchie invizibila
            // intre S si varfurile obsatcolelor
      }

for (i=1;i<nrnod-1;i++)
     for (k=1;k<nrnod-1;k++)
      for (h=k;h<nrnod;h++)
      {a=p[ln[i].ob][ln[i].v];
       b=T;
       c=p[ln[k].ob][ln[k].v];
```

```
        d=p[ln[h].ob][ln[h].v];

        if(intersectie_proprie(a,b,c,d))
         graf[i][j]=0; //elimina muchie invizibila
             // intre T si varfurile obsatcolelor
       }

} //graf_vizibilitate
```

10.3. Temă de laborator

1. Completați algoritmul de mai sus pentru a determina cel mai scurt drum de la S la T în graful de vitibilitate.

Bibliografie

[1] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry, algorithms and applications*, Springer Verlag, 1997, second edition, 2000.

[2] Sergiu Corlat, *Algoritmi şi probleme de geometrie computaţională*, Prut International 2009.

[3] S. Fortune, *A sweepline algorithm for Voronoi diagrams*, Algorithmica **2** (1987), pg. 153-174.

[4] R.L. Graham, *An efficient algorithm for determining the convex hull of a finite planar set*, Information Processing Letters **1** (1972), pg. 132-133.

[5] D. M. Mount - *CMSC 754 Computational Geometry*, Spring 2002, http://www.cs.umd.edu/class/fall2005/cmsc754/lectures.shtml.

[6] J. O'Rourke - *Art gallery theorems and algorithms*, The International Series of Monographs on Computer Science, Oxford University Press, New York, 1987.

[7] R. Sedgewick – *Algorithms*, Addison-Wesley Longman Publishing Co., 1984.